D1.1

First report on software architecture and implementation plan

Stefano Baroni and Stefano de Gironcoli, Pietro Delugas, Andrea Ferretti, Alberto Garcia, Luigi Genovese, Paolo Giannozzi, Anton Kozhevikov, Andrea Marini, Ivan Marri, Pablo Ordejon, Davide Sangalli, and Daniel Wortmann.

Due date of deliverable	31/05/2019 (**month 6**)
Actual submission date	31/05/2019
Final version date	31/05/2019
Revised version date	16/10/2020
Revised version submission date	19/02/2021
Lead beneficiary	SISSA (participant number 2)
Dissemination level	PU - Public

Document information

Project acronym	MAX
Project full title	Materials Design at the Exascale
Research Action Project type	European Centre of Excellence in materials modelling, simulations and design
EC Grant agreement no.	824143
Project starting/end date	01/12/2018 (month 1) / 30/11/2021 (month 36)
Website	http://www.max-centre.eu
Deliverable no.	D1.1

Authors	Stefano Baroni and Stefano de Gironcoli, Pietro Delugas, Andrea Ferretti, Alberto Garcia, Luigi Genovese, Paolo Giannozzi, Anton Kozhevikov, Andrea Marini, Ivan Marri, Pablo Ordejon, Davide Sangalli, Daniel Wortmann.
To be cited as	Baroni et al. (2019): First report on software architecture and implementation plan. Deliverable D1.1 of the H2020 CoE MaX (final version as of 19/02/2021). EC grant agreement no: 824143, SISSA, Trieste, Italy.

Disclaimer

	Change	Author	Note
Change 1	Inserted diagram in Executive Summary	P. Delugas	
Change 2	Extended introduction to section 4	P. Delugas	Highlight examples.
Change 3	Added subsection 4.1	P. Delugas	Define general procedure for library construction
Change 4	Added conclusions section 7	P. Delugas	Summarises important aspect of WP1 which were not part of the SDP

Contents

1 Executive Summary

In the next few years disruptively new hardware architectures will allow massively parallel HPC machines to break the exaflop wall. Porting widely adopted community software for electronic-structure calculations to new and unanticipated hybrid architectures, as well as developing new applications on top of legacy codes, will constitute a considerable challenge. Meeting this challenge will require a substantial shift in the programming paradigm, geared towards the sustainability of a prolonged effort to adapt a software basis of several million code lines to ever changing hardware requirements. Sustainability will require not only that the architecture-specific components of the codes (typically, thousands to few dozens of thousands of lines) are isolated from the bulk of the codes, which could and therefore should remain architecture-agnostic, but also that the effort put into porting few flagship codes will be easily shared amongst codes of a same class, and ideally even across different classes.

The goal of MAX –WP1 is to pave the way to meeting the requirements of *modularity* and *reusability* by refactoring the MAX flagship codes into independent *domain-specific* and *general-purpose* libraries, to be distributed independently of the parent code(s) and where architecture-specific features are isolated and encapsulated into easily portable modules. Such process is schematised in fig. 1.

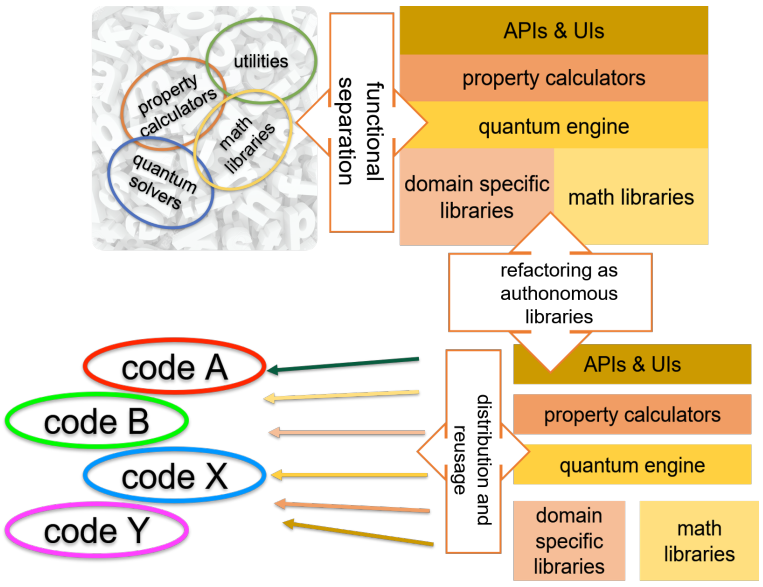

Figure 1: Diagram illustrating the milestones of WP1 work plan on community codes. First stage will be the **functional separation** reorganizing the code into distinct parts dedicated to specific functionalities and accessed via well defined APIs. Second stage will be the realisation of **autonomous libraries**. The libraries will then be available for fast realisation of other codes.

An important side benefit of this programme will be that not only the architecture-specific components, but also the bulk of the codes will be refactored into stand-alone libraries, to be easily shared across different codes, thus allowing one to achieve a considerable economy of scale in the development of efficient and versatile scientific software

in the years to come.

In order to achieve this goal, we proceed along the following path:

1. Defining the criteria and the priorities to be met to refactor legacy code into standalone and shareable public libraries;
2. Identifying the portions of the MAX legacy codes to be encapsulated into public libraries;
3. Defining the criteria and progressively implementing the library APIs that will allow to utilize and share the libraries thus identified;
4. Implementing the libraries and using them starting from their parent codes and progressively extend their use to other codes.

The above programme will be implemented by abiding by the criteria of *autonomy, abstraction, encapsulation, accessibility, data compatibility, flexibility, and documentation*. These guidelines are meant to enforce the adoption of well established good practices of software engineering as well as agree on a general common behaviour of our libraries for what involves interoperability and data management.

The identification of the code functionalities to be refactored, modularised and distributed as public library is inspired by an *optimal utility* criterion: the library will provide tools to deal with concepts and tasks common to as many codes as possible. At a high level: crystal structures, energies, forces, stress, charge densities; and at low level: linear algebra, FFTs, MPI wrappers, error loggers, memory allocation, timers. We will thus cover a wide range of the operations and tasks recurrently performed in electronic structure applications.

Each library will be tagged with a label its indicating its level of maturity: the *proof of concepts* will correspond to the first stages of assembly of a library; *beta releases* will be fully functional, yet not completely validated and benchmarked; *public releases* will be ready for distribution. Although some libraries and their respective APIs have already achieved a satisfactory level of interoperability, for some others a significant development and testing work is still necessary to attain a public release stage. The definition of the APIs and the data structures will be continually updated during the operation of WP1. For most of the libraries the definition of the APIs falls in one of a few paradigmatic cases. As representative examples of the issues posed by the definition a of public API and of the solutions that we plan to adopt, we discuss in detail the common general APIs of: the `FFTXlib` library, giving access to FFT operations and manage 3D data grids; the `LAXlib` library for parallel linear algebra, providing transparent access to the most widely-used linear-algebra operations and allowing one to manage distributed or offloaded matrices and vectors data; the `PSolver` from BIGDFT, which provides an API template to instantiate operators acting on 3D data grids in a general and transparent way. Finally, we will illustrate the common general API that will allow to instantiate, initialize, and access, quantum engines as objects inside third party applications. In order to achieve similar functionalities, appropriate *hooks* will be implemented in the `SIRIUS` DSSDP.

Each code consortium has organized a development plan for the selected set of functionalities they will provide. The libraries will be distributed collectively via the MAX gitLab repository, and the consortia will collaborate within the WP as a whole to give

them a coherent and interoperable structure. To this end, a few inter-consortium working groups have been organized, and other will be organized in the future when need arises. For the time being, the following groups are up and running:

1. The *FFT* work-group will take care to implement, and test the common API for the FFT libraries.
2. The *Parallel linear algebra* work-group will provide examples and benchmarks for the usage of the `LAXlib` libraries for performing parallel linear algebra.
3. The *Quantum Engine Interfaces* work-group will implement reusable codes for the sampling and evolution of atomic configurations in interaction with generic quantum engines. This work-group will also take charge to define, test, and implement the common `API` to access the internal functionalities of the quantum engines.
4. The *Symmetry* work-group will implement common tools and libraries to detect and exploit space symmetries in materials and to automatically determine sets of k points for Brillouin-zone integration.
5. The *Code documentation* work-group will take care to produce an organic documentation of the software platform delivered by WP1 and the API definition.

2 Introduction

In WP1 the architecture of the MAX flagship codes will be organized to make them ready to run on pre-exascale machines, scheduled to be deployed in Europe in the next few years. Our software development aims to:

- Deliver a release of the MAX flagship codes ready to run on pre-exascale machines by the completion of the program, and provide exascale-ready software components to be adopted by other quantum-simulation community codes, which are not members of the MAX consortium;
- Design and implement a software development model, based on inter-code and inter-architecture portability, that will allow us to keep pace with the swift and unanticipated turns that hardware architectural evolution [1] will make in the years to come.

The development activity in WP1 is interwoven with those of WP2 and WP3 and will exploit the architecture-specific and performance-optimized code components delivered by WP4. Leveraging of effort of WP8 for the organization of *hackathons* and similar events the development in the WP will be open and receptive for the activities outside the MAX project. In particular it is worth to mention the interaction that WP1 will have with the ESL [1] initiative of CECAM. The philosophy of "open innovation" espoused by ESL, based in the development of a suite of modules for electronic-structure calculations, is very similar to our own. The ELSI project [2] is pursuing similar ends in the field of electronic structure solvers. It should be noted that three MAX researchers sit in the ESL Steering Committee, and also collaborate with the ELSI project.

We have organised the WP in two interdependent tasks:

T1.1: identifying key components in the MAX codes and defining the criteria of optimal portability to other codes and different architectures; refactoring the MAX codes accordingly; packaging and releasing the stand alone libraries;

T1.2: identifying and implementing methodologies for the integration of heterogeneous components from different software sources (codes/libraries) into an exascale-ready interoperability platform; in doing this we will pursue two approaches:

- proceeding top-down we will provide access to functionality of well-structured, exascale-ready software without extracting any individual component;
  - the `SIRIUS` platform will be integrated with *hooks* which will allow to access the platform functionality at various levels of granularity, from quantum engine to its underlying library components;
  - the internal functionalities of MAX quantum engines and other applications will be made readily available by exposing public APIs with multi-language bindings.
- proceeding bottom-up we will integrate the high-performance specific components extracted from the flagship codes at various levels of granularity; we

[1] See the D4.1 document.

will implement demonstrations of this concept of reuse and provide a complete documentation.

The extraction of the key components from the MAX codes as well the implementation of the *hooks* in the SIRIUS platform will be worked out mainly within each code consortium and distributed via the gitLab repositories. The detailed list of the planned activities around this side is discussed in detail in section 4.

When *proof of concept* or mature version of the libraries is ready, the integration and the improvement of the APIs interoperability will start. This part of the development will be done collaboratively by all the participants to MAX and possibly outside. In collaboration with WP8 we will organize schools and *hackathon* events to give publicity and impulse to these collaborative actions.

This collaborative development will be organised in inter-code work-groups dedicated to specific code integration, interfaces and mini-apps implementation, code portability testing , documentation redaction.

The following sections will present in detail the various aspects of our plan. In section 3 we will illustrate the general software architecture that we intend to realize and the interoperability criteria that we are going to adopt. In section 4 we provide a description of the modularization process and of the libraries and development tool that we plan to refactor and distribute. In section 5 we present important examples of the most important features of our set of APIs. In the last section 6 we describe the plan and the organization of our collaborative actions; finally in the conclusions we discuss briefly other important points concerning the activities of WP1.

3 Library factorization and interoperability criteria

The basic points of the software architecture we are planning are: the **separation** of the architecture-specific functionalities from the main architecture-agnostic code basis, planned in such a way as to enhance the maintainability and portability of our code; the reorganization of the most relevant **data structures** using data-types whose initialization, allocation, access and destruction is managed by **general interfaces** designed in order to simplify the usage of accelerated architectures and openMP multithreading. This work will benefit from strong interchange with activities of WP4 and form the basis of the development effort of WP2 in which the separation of functionalities we describe here will enable the implementation of performance-portable solutions.

To extend the reuse of the work done by refactoring the flagship codes, we also plan to package and distribute a selected set of functionalities as stand-alone autonomous libraries. To refer to these functionalities in the following, we will use the term "modules", to be understood in a broader sense than in context like `Fortran` or `Python` programming languages.

The identification of modules suitable for such a distribution will be made using a criterion of **optimal utility**: the modules' exposed APIs should deal with concepts and tasks common to as many codes as possible (e.g. at a high level: crystal structures, energies, forces, stress, charge densities, etc, and at low level: parallel linear algebra, FFTs, MPI wrappers, error loggers, memory allocation, timers, etc), and cover a wide range of operations recurrently used in electronic structure applications. Some of these tasks, in order

to be efficiently used in exascale machines, are going to require an architecture-specific implementation.

To enhance the effective interoperability of our libraries we will fulfil the following criteria:

- **autonomy**: Being distributed in a repository independent of that of the code from which it originates, with a clear list of dependencies and a stand-alone standard (cmake, autoconf) build-system that enables the compilation on a variety of platforms;
- **abstraction:** the implementation should avoid any code specificity so as to be reusable as widely as possible; the functionality should be implemented abstracting as much as possible from any particular case, so as to maximize the number of applicable tasks;
- **encapsulation:** the behaviour of all the subroutines composing the libraries should depend only on data supplied in input with no usage of any global variable; routines will pass and share common complex data using structured data-type arguments used as *descriptors* or *state vectors*;
- **accessibility:** the API should provide methods to initialize and update the datatypes
- **data compatibility:** data types used to define arguments to routines should be interoperable with Fortran and other most common languages *e.g.* `C/C++` and `python`.
- **flexibility:** the libraries should be thread-safe and for those implementing more compute-intensive functionalities (FFT, linear algebra, eigensolvers) the API should handle the usage of accelerated architectures.
- **documentation**: a reference documentation website, in form of a *library user*'s documentation, that specifies:
  - The scope of the library, functionalities and its relation with electronic structure calculations;
  - The API, the main library datatypes and some examples of their usage;
  - The level of maturity of the package, chosen between proof-of-concept, alpha-release, release candidate, production version;
  - A list of known problems and issues, including projects for future functionalities;
  - capability of usage in a massively parallel environment, and possibly in a pre-exascale supercomputer.

The above principles will be used only when they appear feasible and sensible; as a trivial example: it would be clearly useless to implement `python` or `C/C++` interfaces for the I/O libraries as long as standard data formats are used. The possibility and necessity to follow these criteria will be evaluated case by case and reported in the D1.3 document together with any other evolution of our software architecture that will occur during the development and code gluing activities.

Depending on the size of these libraries we will also make it possible to insert them in third party packages as sub-modules to be configured and compiled together with the other source code. In this case we will document how configure these sub-modules.

For the case of independent compilation we will aim whenever possible to provide interface modules or include files to allow `C` compatibility and avoid compiler interdependence in `Fortran`.

An essential part of our architecture will be the definition and implementation of a general API to initialize, run and query the quantum engines so as to provide access to the most relevant functionalities of our –and in perspective any other– quantum engines in library mode. The interface will be based on common and flexible data-structures satisfying the criteria already listed above, a preliminary description of the API is provided in section 5.4. This work group will decide on the final details of these APIs during the first 18 months, the results of this exploratory activity will be reported in the D1.3 document. The description and the documentation of the APIs will be distributed in MAX repository on `gitLab`.[2]

4 Identification of domain-specific libraries and modules

The realisation of the MAX library bundle is, as outlined in fig. 1, part of the reorganization of the community codes aimed at separating the code parts that implement specific functionalities from the main code base. The separation will allow for the autonomous development and maintenance of each of these code parts, making the overall maintenance of electronic structure community codes more sustainable and portable.

In the earlier part of the operation period, WP1 has worked at the identification of specific functionalities to be extracted from the main codes and refactored as libraries. The choice of such functionalities has also been inspired by the objective of being able to provide an effective set of libraries that can be reused for developing more computationally efficient applications in the electronic structure field.

As already anticipated we have followed an *optimal utility* criterion in the selection of the functionalities, redesigning and implementing each of these functions in view of a broad general usage. We have selected functionalities that are either frequently used throughout the codes, or represent heavy computational kernels that is important to isolate and optimise, or constitute well-known and used building blocks of electronic structure codes. The functionalities selected so far are:

- Formatted and hierarchical I/O (XML, YAML, HDF5)
- APIs and data-structure for timing and profiling
- Error handling and logging
- Abstraction layer for MPI initialisation and MPI interface access
- General Mathematical libraries
- Domain Specific libraries performing high-level functions specific of the electronic-structure field.

[2] `https://gitlab.com/max-centre`

All of these functionalities are crucial for the flexibility, integration and portability of the codes.

For example, the interfaces for the formatted hierarchical I/O allow one to easily adapt the data format of electronic structure codes to those of external applications and databases and will also streamline the adoption of common data formats among our electronic structure domain specific libraries. An example of the usage of these formats is libPSML in SIESTA, that has been refactored as a completely autonomous library. The library uses a general purpose XML writer/parser library (xmlf90), maintained by the same SIESTA group, to write and read pseudopotentials written accordingly to a well specified XML scheme (PSML). The usage of more XML schemes in other codes will be streamlined by a Python tool added to the MAX bundle by the QUANTUM ESPRESSO group that automatically generates Fortran schema-specific writing/parsing routines, only taking a schema XSD file as input.

An example of a library that gives access to low-level functionalities is `UtilXlib`, developed by the QUANTUM ESPRESSO group. This library provides a wide set of interfaces for the initialisation and usage of MPI parallelism, error handling, timing and profiling. These interfaces have been defined and implemented together with the experts from the computing centres that within WP4 will adapt and tune them to different architectures.

The design and implementation of the mathematical libraries will be done in tight interaction with WP2 and WP4, which tackle the task of implementing efficient architecture-agnostic interfaces for these compute-intensive parts of the codes. Examples of these mathematical functionalities are the 3D Fourier transforms performed by `FFTXlib`, prepared and maintained by the QUANTUM ESPRESSO group, and the `SpFFT` library written in `C++` by the `SIRIUS` group. Similar actions have also been taken for distributed linear algebra (`LAXlib`) and other general mathematical functionalities. In all these cases we are working with WP2 on the API design in order to improve the performance portability of the quantum-engine codes. Some example on the work of API design is given in section 5.

Together with these general functionalities, WP1 will work also on the implementation of libraries performing tasks which are more specific to electronic structure computation. These libraries will help the work of WP3 on algorithmic improvements. Moreover, the newly developed and implemented algorithms will be made accessible to third party developers as part of the MAX libraries.

After subsection 4.1 that describes the general path followed for the preparation of the libraries, with the definition of the milestones and maturity stages, the rest of the section is dedicated to the description of the libraries, their functionalities and possible cross-dependencies with other packages. As the work on the libraries will be mainly done within each code-consortium, in parallel with the refactoring of the flagship codes, we present the libraries per provenance code.

4.1 Preparation of the libraries, milestones and maturity stages

The code parts that are going to be reorganised as libraries present different starting degrees of modularisation. Depending upon this, the detailed refactoring plans may significantly differ. In order to have a common understanding of the completion status of each library, we define on general grounds the subsequent incremental refactoring steps

undertaken for each of the libraries:

- a first stage where we simply identify the code parts and isolate them in distinct directories;
- we rewrite the interface to expand the scope and the usability of the isolated functions, following as much as possible the criteria exposed in section 3 and, when needed, implementing missing functionalities or integrating them in a more general form;
- we refactor the main code adopting the new interfaces; this is the first milestone, the so called **proof of concept** stage ;
- we complete the encapsulation of the data structures in the proof of concept, and if needed, integrate the API with initialisation routines;
- we separate the module from the original code, providing it with an autonomous build system and effective methods for exposing the interfaces and the data structures to the linking codes; the library reaches now the second milestone that we refer to as **beta** stage and it is ready to be linked as an autonomous object to third external codes;
- we work at the refinement, bug fixing, and improvement of the interface until we reach the **production** stage of the library.

As libraries reach the beta stage we will start to experiment their usage outside the original codes so at to assess the flexibility and interoperability of the interfaces. A first map of the reuse plan is presented in fig. 2

In the set of libraries planned at this stage there is inevitably some amount of redundancy, as for example in the case of the I/O management libraries, error handling, and others. When the libraries are delivered we will evaluate the necessity, opportunity and feasibility of reducing such redundancies, versus keeping them when they represent a resource in term of versatility and resilience. We will report about this issue in the D1.3 plan update at month 18.

4.2 BigDFT code

- **FUTILE library:** The FUTILE project (Fortran Utilities for the Treatment of Innermost Level of Executables) is a set of modules and wrapper that encapsulate the most common low-level operations of a Fortran code. It provides wrappers and controls for (log)file dumping, string handling, input file parsing, dynamic memory allocations, profiling, error handling as well as MPI interfaces and Linear algebra wrappers. It also implements advanced data storage objects like linked lists and trees, and provides their bindings to python dictionaries as well as iterators. This package is meant to simplify the work of Fortran code developers as its APIs are inspired from Fortran approach. Particular attention is paid in not downgrading the performance of the upper level subprograms. The API of FUTILE project is defined and almost stabilised at the time of the writing of the present deliverable. Its documentation is now in its stabilisation phase and it can be found at the

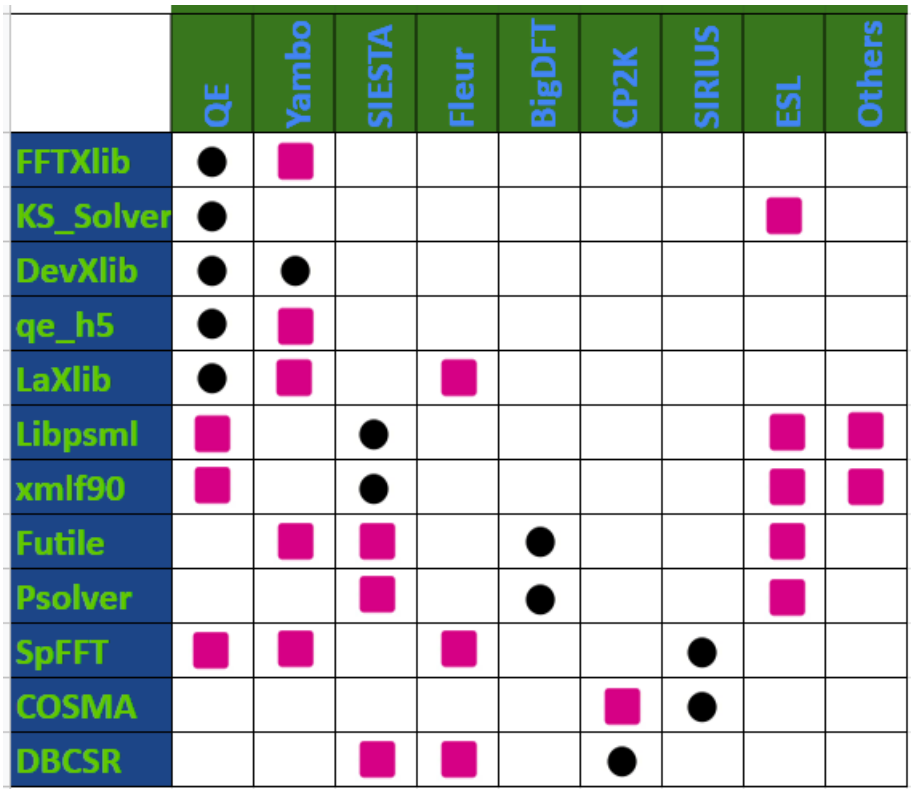

Figure 2: Map illustrating the reuse of MAX libraries within the consortium codes and outside. The black dot indicates the original code and the purple squares the codes that plan to reuse the library.

URL https://l_sim.gitlab.io/futile/. Such API specifications are already in use in some of the high-level routines of the other subpackages of the BigDFT consortium, e.g. PSolver, CheSS.

- **PSolver:** This package features a real-space based solver employing interpolating scaling functions (ISF) for the solution of the Poisson Equation in vacuum and in continuum solvents. ISF provide features of flexibility and precision that make this package well suited for a integration in various Electronic Structure codes. It also provides GPU acceleration and parallelization scheme that make its usage suitable for calculations of the action of the Fock Operator, that is of great interest in the context of Hybrid functionals calculations. Like in the case of FUTILE library, the API of the PSolver library has been identified and it is under documentation at the URL https://l_sim.gitlab.io/psolver/.

- **atlab:** This library deals with common operations which have to be performed on the atoms in the context of a electronic structure code. It abstracts the representation of simulation domain, iterator on real-space points, atomic structure and related concepts (I/O of voluminous files, handling of atomic basis functions, symmetry operations, just to name a few), and it is also meant to provide a software development platform to define ionic movement operations *prior* to the actual specification details of the Quantum Engine. By using `atlab` API the developer will be able to separate the concerns related to handling of ionic movements from the internal representation of the atomic structure provided by the employed electronic structure code, in a similar spirit of the Atomic Simulation Environment Python package (see https://wiki.fysik.dtu.dk/ase/). The `atlab` API is designed to preserve the massively parallel spirit of the internal code operations, and also provides lower-level handling routines which are typical to a electronic structure code, abstracting them from the computational basis set employed in the calculations.

- **libconv:** The BigDFT code employs wavelets as a internal computational basis. Some of the operations involving wavelets may be written in terms of convolutions with short, separable filters. Some of these operations are formally similar to real space treatments like finite-differences calculations. For this reason, the library `libconv` will be released, such as to define an API that might implement the action of a generic convolution in a HPC framework. Such a library is conceived and written thanks to the BOAST meta-programming engine (see https://www.rubydoc.info/github/Nanosim-LIG/boast/master and [3]), which is able to perform source-to-source optimisation and abstracts the convolution generations. Such a programming paradigm is also of great utility in the context of autotuning and co-design which will be treated in WP2 and WP4 respectively.

- **PyBigDFT:** This package is a collection of Python Modules that are conceived for pre- and post- processing of BigDFT input and output files. Such modules are supposed to enhance the BigDFT experience by high-level approach. Also, calculators and workflows are supposed to be created and inspected with modules

of the PyBigDFT package. This package is conceived as a set of Python modules to manipulate complex simulation setups in a HPC framework.

- **bundler:** Such package is defined from a fork of the `Jhbuild` package,[3] that has been conceived in the context of GNOME developers consortium. When creating a suite code that is made by a collection of various libraries released independently, the problem of linking and compiling together the entire suite might become cumbersome and time-consuming, especially for non-expert users and satellite developers. The `bundler` package provides a set of solutions which try to address this problem. Like the `jhbuild` package, it is able to compile libraries with different build systems and configuring options. However, it has been particularly tailored for compilations in supercomputers frontends and HPC architectures, which makes its usage particularly interesting for high performance codes in computational science.

- **sphinx-fortran:** This project is a fork of the sphinx-fortran package,[4] which has been written in order to use the `sphinx` package to describe package documentations. Such library provides an extension – alternative to other solutions like FORD and Doxygen – which will be of interest for fortran source codes which might be connected to other high-level programming languages like python. With this package fortan API might be exposed (and referenced) together with other programming languages.

4.3 CP2K code

CP2K is a quantum chemistry and solid state physics software package that can perform atomistic simulations of solid state, liquid, molecular, periodic, material, crystal, and biological systems. CP2K provides a general framework for different modelling methods such as DFT using the mixed Gaussian and plane waves approaches GPW and GAPW. Supported theory levels include DFTB, LDA, GGA, MP2, RPA, semi-empirical methods (AM1, PM3, PM6, RM1, MNDO), and classical force fields (AMBER, CHARMM). CP2K can do simulations of molecular dynamics, metadynamics, Monte Carlo, Ehrenfest dynamics, vibrational analysis, core level spectroscopy, energy minimization, and transition state optimization using NEB or dimer method. CP2K is written in Fortran 2008 and can be run efficiently in parallel using a combination of multi-threading, MPI, and CUDA. It is freely available under the GPL license. It is therefore easy to give the code a try, and to make modifications as needed.

- **libDBCSR:** The core functionality of the linear scaling electronic structure method of CP2K code is provided by the `libDBCSR` — a sparse matrix-matrix multiplication library. It is developed as part of CP2K, but can be used as a standalone library. It is hosted on `https://github.com/cp2k/dbcsr` and is already available for integration in other projects. The library has been tested to run on CPUs and hybrid architectures (CPUs+GPUs) on Piz Daint supercomputer at CSCS.

  The following developments in `libDBCSR` have been finished recently:

[3] `https://developer.gnome.org/jhbuild/`
[4] `https://sphinx-fortran.readthedocs.io/en/latest/`

  - just-in-time (JIT) compilation of matrix-matrix multiplication kernels for a given (m,n,k)-triplet
  - machine learning (ML) prediction of optimal matrix-matrix multiplication kernel parameters
  - migration to the CMake build system

The following tasks are planned next:

  - Tuning the performance of the library for the Power9 + V100 NVIDIA GPU architecture (Summit supercomputer at ORNL).
  - Porting `libDBCSR` to AMD GPU cards using ROCm platform
  - Transfer learning: using the ML-optimized kernels for NVIDIA P100 GPU cards to generate optimal parameters for the new GPU hardware.

4.4 FLEUR code

The FLEUR code is an implementation of the full-potential linearized augmented plane-wave method. In contrast to the other flagship DFT codes of the consortium it is an all-electron code not using pseudopotentials. While this introduces some unique challenges, we also identified several parts of the code of more general interest that have been insulated and will be turned into stand-alone libraries:

- **Basic utilities (juDFT-Library):** Significant parts of the basic setup, IO, timing and error handling performed in FLEUR a by now separated from the main code and collected in the juDFT-Library. In particular this includes code for interfacing FLEUR with the libxml2 standard library for XML input and utilities for HDF5-IO. We plan to continue the refactoring of the code such that the interface to these functionality becomes more exposed. In a later step the possibility to merge our functionality with that provided in similar libraries of other flagship-codes like UtilXlib of QE or FUTILE of BigDFT will be explored.

- **Linear algebra (FLEUR-LA):** As the solution of a dense-matrix generalized Hermitian eigenvalue problem is usually the most expensive operation to be performed in a standard FLEUR calculation, we already implemented interfaces to a the most relevant HPC libraries available for this task. This code also has a clear interface for non-distributed matrices as well as for matrices distributed on multiple MPI tasks. It can be viewed as a stand-alone module. As it is similar to the functionality provided in LaXlib of QE we will aim at a convergence here (See API section on linear algebra below). Besides the direct solvers routinely applied to the problem, we also have an interface to an iterative solver and plan to investigate its range of applicability exploiting the particular features of the DFT self-consistency process.

- **Matrix operations needed for hybrid functionals (LAPWlib):** While FLEUR is able to evaluate hybrid functionals, such calculations require significant more CPU time than more standard DFT functionals due to the more complex algorithm but also because of the little optimized code. Here we plan construct a highly efficient library of basic matrix operations in the LAPW basis needed to speed up these

calculations. While we will start having the application to hybrid functional as the focus of interest, these operations will also be useful for the evaluation of other properties or other methods (for example the reduced-density matrix functional theory (RDMFT)).

- **IO functionality for complex setup datatypes (IO-t):** `FLEUR` uses many datatypes for storing the multitude of parameters describing the basic calculation setup. In order to facilitate a more flexible program flow including high-level scripting capabilities as needed on future heterogeneous or modular supercomputers we plan to augment these types by storage functionality providing a unified possibility to perform IO operations on these types and to distribute them efficiently.

4.5 QUANTUM ESPRESSO code

QUANTUM ESPRESSO is a suite of electronic structure codes based on pseudopotential an plane waves. The suite comprises `pw.x` for standard electronic structure calculations, `cp.x` for Car-Parrinello molecular dynamics and a suite of applications based on DFPT and TDDFPT to compute phonons and optical spectra.

The set of libraries and utilities extracted from QUANTUM ESPRESSO include the general functionality and compute intensive mathematical layer adopted through all the suite as well as the `KS_solver` collection of eigensolvers used in `pw.x` and the `LRlib` library which is meant to provide general abstract access to the whole operational apparatus used in DFPT and TDDFPT applications.

- **UtilXlib:** this is a library of interconnected utilities which provides:
  - an API for the initialization and management of MPI and OpenMP parallelism; next versions will also include utilities needed to exploit accelerated architectures and the developments on adaptive parallelism planned in WP3;
  - error handling routines which we plan to integrate in the next versions with a more general logging utility which in the next versions will also allow to print out the log to arbitrary streams and will adopt a standard `YAML` format;
  - timing routines, in the planned released we will add to the library the interfaces to more timing and profiling utilities.
- **I/O management:**
  - The `Fortran` API used inside QUANTUM ESPRESSO to manage HDF5 I/O will be released as a library (`qeh5`). We plan to improve the management of parallel HDF5, data compression and mixed precision usage. To enhance the portability we plan to refactor future versions of the library so as they access directly the `C` API of HDF5, avoiding the use of `F2003` interface, which currently forces to use an HDF5 library built with the same compiler used to compile the calling codes;
  - The toolchain used in the development of QUANTUM ESPRESSO to build the XML I/O routines and data-types will be released publicly. This tool based on `Python` and `jinja2` templates allows, starting from an XML

schema, to generate Fortran data-types and routines for reading, writing and initialization.

- **FFTXlib:** A working group constituted by FFTXlib (QUANTUM ESPRESSO) and FFT3D (SIRIUS) developers has defined a unified API transparent to the particular distribution of 3D grids over tasks. This interface will be implemented and adopted for FFTXlib. In collaboration with WP4 the library will be adapted to pre-exascale upcoming architectures.

- **LAXlib:** this is a parallel linear algebra front- end, it is already used in all QUANTUM ESPRESSO suite to provide transparent access to high performance specific libraries. In the context of WP2 the integration inside FLEUR and YAMBO will be tested. This integration will provide useful input for the preparation of a common general API for parallel linear algebra (see section 5.2). We will implement the new common general API as soon as it will be *production* ready. In collaboration with WP4 the library will be incrementally adapted to manage parallel linear algebra in emerging HPC architectures.

- **LRlib:** an important effort is planned on this library which performs a variety of tasks connected with DFPT. Objectives of this effort are:

  - Full API definition and documentation;
  - Performance improvement removing inefficiencies, expanding OpenMP parallelism and preparing a GPU ready version;

- **KS_solvers:** This library which contains the iterative eigen-solvers used in QUANTUM ESPRESSO and other electronic structure codes will contain the improvements and enhancements planned in WP3 regarding the development of more robust algorithms and of adaptive schemes for diagonalisation and mixing. Portability will be improved by the addition of RCI interfaces and more examples and tests, on this side it is worth to mention an important collaboration with the ELSI infrastructure.

- **UPF_pseudolib:** a library for handling **pseudo-potentials** (PPs) is in preparation in collaboration with YAMBO consortium. The library will allow one to:

  - read and extract data from pseudo-potential files;
  - perform radial and 3D initialisation of the PP data.
  - evaluate the local and non-local contributions of the pseudopotentials to the KS Hamiltonian (including scalar products of wavefunctions and PP projectors).

- **XCfunc_Xlib:** definition of a library for the portable handling of exchange-correlation (XC) functionals, including (full and range-separated) hybrids and van der Waals (vdW) functionals. The library will be developed in collaboration with the YAMBO consortium.

4.6 SIESTA code

SIESTA's defining feature is the use of strictly localized pseudo-atomic orbitals (PAOs) as basis set. This makes it very efficient for large systems, and also sets it apart from plane-wave codes regarding the internal methodology. The Hamiltonian and overlap matrices are sparse, allowing for the use of specializzed solvers and also leading to a linear-scaling operation count for their setup.

- The **GridXC library** deals with the computation of the exchange and correlation energies and potentials in relevant real-space grids: parallelepipedic for 3D periodic systems (including artificial periodicity) and spherically symmetric for atomic-like systems. It was the original vehicle for efficient implementation of vdW functionals. Now it can also use the density functionals provided by the libxc library. The library is quite mature, but some extra work is planned:
  - Exposing more functionality to clients (e.g., a load-balancer for grid-point distribution)
  - Offering more choices for parallel-redistribution routines
  - Replacing internal fft routines by calls to FFTW/PFFT libraries in the vdW section.

  This library is directly usable by any code, in particular those in the MAX consortium.

- The **LibPSML library** is the main piece of the ecosystem of tools to handle pseudopotentials in the PSML format (see http://esl.cecam.org/PSML). As of now, it can handle norm-conserving pseudopotentials and offers a Fortran interface. More interfaces (C, Python), associated tools (e.g., conversion to and from UPF2), and a possible extension to ultra-soft pseudopotentials and PAW datasets are planned.

  This library can be useful for most codes in MAX and beyond, with the obvious exception of all-electron codes such as Fleur.

- The **interface to the ELSI library** [2] in SIESTA is quite streamlined, as ELSI natively supports mechanisms for passing to solvers sparse H and S matrices, and returning the density-matrix, all in the SIESTA format. Currently, the library offers direct solvers (ELPA) and specialized solvers (PEXSI, OMM, density-matrix purification) which are most useful for LCAO-type codes such as SIESTA, but interfaces to iterative solvers, of interest for plane-wave codes, are already in advanced development.

  The SIESTA-ELSI interface can be abstracted some more to turn it into a meta-package that could be plugged in similar codes.

- A module for **neighbour search in $O(N)$ operations** can be extracted from SIESTA and offered as an independent library.

- The technology for using and **embedded Lua interpreter** for internal scripting (based on a number of submodules: the Fortran-Lua bridge, dictionary modules, etc) has already proven itself in a number of applications in SIESTA. The individual components can be further packaged to be useful in any code.

- The **FDF** (input file processing), and **xmlf90** (generation and parsing of XML in modern Fortran) libraries are quite mature (and already part of the ESL [1] bundle).

4.7 YAMBO code

YAMBO is a scientific code implementing Many-Body Perturbation Theory methods both at equilibrium and out-of-equilibrium. It uses DFT Kohn–Sham states as reference basis to calculate, ab–initio, several ground–state and excited–state observables. YAMBO encodes several widely used techniques, such as the GW approximation for the electronic self–energy or the Bethe–Salpeter equation (BSE) to account for excitonic effects in the optical absorption. As YAMBO deals with excited states, in addition to some of the basic tools used in ground–state codes (like FFT), it also uses several specific algorithms designed to work on very large matrices or in large Fock spaces and to handle massive Input/Output operations. In addition YAMBO adopts a peculiar user interface that allows the code to be entirely controlled from the command line.

The above mentioned features of YAMBO have been packed in a series of modules that we aim at organising in such a way to be distributed in the form of agnostic libraries. Each library will be provided with an interface and examples and the source will be hosted on a dedicated and open GIT repository. The libraries are:

- **Driver_Ylib:** a library that can be used to equip any code with a simple and intuitive command line tool. The library will allow one to:
  - delegate specific actions to the command line;
  - easily interact with external scripting tools;
  - easily support newly added run–levels and features.
- **CoulCut_Ylib:** a library to wrap and distribute the multiple techniques proposed in the literature and implemented in QUANTUM ESPRESSO ad YAMBO to deal with the truncation of the Coulomb potential and the regularization of integrals and expressions involving its long range divergence. This is particularly relevant since these expressions are ubiquitous in electronic structure methods (ranging from electrostatics in periodic boundary conditions to hybrid functionals and many-body perturbation theory). The library will allow to:
  - provide a consistent treatment of the different steps needed for any specific calculation;
  - provide generalized procedures that work also in particularly severe cases (e.g. GW on top of DFT data computed using hybrid functionals);
  - complement the Coulomb cutoff definition with specific regularisation tools to handle divergences appearing in low–dimensional systems.
- **LA_Ylib:**. YAMBO implements its own interface to several linear algebra libraries (such as Lapack, ScaLapack, PETSC, SLEPC) together with a general purpose layer to handle the different parallel data distributions required by the different libraries. We plan to base the interface on isolated modules and routines so to modularise it. The library will allow one to:

  - drive linear–algebra operations on arbitrary large matrices by using direct and iterative algorithms provided by ScaLapack, PETSC, and SLEPC;
  - provide a series of tools to transform distributed matrices from one parallel structure to another. Indeed **LA_Ylib** will support several structures: BLACS, PETSC, line-by-line parallel distribution, BSE structure. The tools in the **LA_Ylib** will allow to transform one structure to another without allocating the entire matrix.

- **IO_Ylib:**. The YAMBO I/O is one of the most advanced and performing parts of the code. This is due to the fact that several quantities are written by YAMBO at running time, and most of them can be very large. This implies that, in order to be performing, YAMBO I/O is made, at the very low level, by using NetCDF and HDF5 instructions. The **IO_Ylib:** library will allow one to:
  - define a series of agnostic procedures to open, close, access, remove, rename the I/O files, treated as generalised databases;
  - provide support to any kind of I/O as the actual write/read of the data is localised in very few specific routines.

4.8 SIRIUS Software Development Platform

SIRIUS is a domain-specific software development platform (DSSDP) for electronic structure calculations designed and implemented at ETH Zurich. The platform supports both plane-wave pseudopotential and augmented plane-wave full potential methods and is designed from ground-up to run on GPU-enhanced hybrid architectures. The SIRIUS quantum engine has been successfully interfaced with the property calculators and I/O layers of QUANTUM ESPRESSO. SIRIUS has linear-algebra and FFT submodules, both of which are GPU-accelerated, that can be shared with the MAX codes using a common set of APIs where needed/appropriate. Hooks will be provided to access SIRIUS internal data and functionalities from third-party applications. including HTC managers and code-gluing environments. Moreover the work is in progress to integrate the SIRIUS quantum engine into the CP2K code.

5 APIs

The application of our interoperability criteria relies heavily on the construction of effective APIs designed for a architecture agnostic access to low level functionalities as well as for accessing to high level functionalities of libraries or of fully instantiated quantum engines (our flagship codes as well as SIRIUS DSSDP) abstracting from their specific implementation. The conception and evolution of the APIs will thus require in many cases an important testing phase, as well as a continuous update to necessities which may emerge by the application to new hardware and software specifications. For this reason the WP1 APIs will be continuously updated during the progress of the project. The whole set of APIs will be provided via the MAX repository, and continuously updated.

In these early definitions of the APIs we have singled out the main difficulties that the definition of interoperable interfaces and data structures may present in our field, and we have also agreed on some general solutions to adopt for similar cases. To illustrate

Expected library readiness up to M18

Library	Group	Expected release	Month M6	Month M12	Month M18
FUTILE	BIGDFT	M12	Beta	Production	—
PSolver	BIGDFT	M12	Production	—	—
atlab	BIGDFT	M36	P.o.C.	P.o.C.	P.o.C.
libconv	BIGDFT	M24	P.o.C.	Beta	Production
bundler	BIGDFT	M24	P.o.C.	Beta	Production
PyBigDFT	BIGDFT	M24	Beta	Beta	Production
sphinx-fortran	BIGDFT	M24	P.o.C.	Beta	Beta
juDFT	FLEUR	M24	P.o.C.	Beta	Production
FLEUR-LA	FLEUR	M24	P.o.C	Beta	Production
LAPWlib	FLEUR	M36	P.o.C.	P.o.C.	Beta
IO-t	FLEUR	M36	P.o.C	P.o.C.	P.o.C.
qeh5	Q. ESPRESSO	M12	Beta	Production	Production
xmltool	Q. ESPRESSO	M12	Production	Production	Production
UtilXlib	Q. ESPRESSO	M24	P.o.C.	Beta	Production
FFTXlib	Q. ESPRESSO	M24	Beta	Beta	Production
LaXlib	Q. ESPRESSO	M24	Beta	Beta	Production
KS_solvers	Q. ESPRESSO	M24	P.o.C.	P.o.C.	Beta
LRlib	Q. ESPRESSO	M36	P.o.C.	P.o.C.	P.o.C
UPF_lib	Q. ESPRESSO YAMBO	M36	P.o.C	P.o.C.	Beta
XCfunc_Xlib	Q. ESPRESSO YAMBO	M36	P.o.C.	Beta	Beta
Driver_Ylib	YAMBO	M24	P.o.C.	Beta	Production
ColCut_Ylib	YAMBO	M24	P.o.C.	Beta	Production
LA_Ylib	YAMBO	M24	P.o.C.	Beta	Production
IO_Ylib	YAMBO	M24	P.o.C.	Beta	Production
GridXC	SIESTA	M24	Beta	Beta	Production
libPSML	SIESTA	M24	Beta	Beta	Production
ELSI-interface	SIESTA	M24	Beta	Production	—
LibNeigh	SIESTA	M24	P.o.C.	Beta	Production
Lua scripting	SIESTA	M24	P.o.C.	Beta	Production
libFDF	SIESTA	M24	Beta	Production	—
xmlf90	SIESTA	M12	Production	—	—
libDBCSR	CP2K	—	Production	—	—

Table 1: Present and Expected Level of Maturity of the WP1 libraries during the first 18 months. **P.o.C. :** *Proof of concept* version, **BETA:** release candidate, **Production:** interoperable library ready for release.

these general design issues and their solutions we describe in this section the work-plan and the API definition for: the `FFTXlib` library, giving access to FFT operations and manage 3D data grids; the `LAXlib` library for parallel linear algebra, providing transparent access to the most widely-used linear-algebra operations and allowing one to manage distributed or offloaded matrices and vectors data; the `PSolver` from BIGDFT, which provides an API template to instantiate operators acting on 3D data grids in a general and transparent way. Finally, we will illustrate the common general API that will allow the user to instantiate, initialize, and access, quantum engines as objects inside third party applications.

5.1 FFT common API

Fast, distributed and accelerated FFT library for the transformation of the *subset* of plane-waves (a sphere of plane-wave coefficients in the reciprocal space) are not yet fully available. Several open-source FFT libraries exist, for example FFTW, accFFT, PFFT, but none of them fulfils all of the above-mentioned criteria.
Minimal requirements for the FFT library:

- sequential and parallel transforms
- CPU and GPU back-ends
- handling of CPU and GPU pointers
- handling of the reduced (by inversion symmetry) set of G-vectors; transformation of real functions from a reduced set of plane-wave coefficients
- simultaneous transformation of two real functions (Gamma-point case)
- transformation of the “sphere“ and “full box“ of plane-wave coefficients
- handling of the large FFT boxes with up to 4000 points along each of the dimensions

Optional requirements:

- transformation of arbitrary list of plane-wave coefficients
- explicit complex-to-real transformations

Optional requirements are not strictly necessary but can add an extra benefit to the library.
The following assumptions are made:

- 1D/2D CPU FFT implementation is available though MKL or FFTW3; no custom 1D/2D FFT kernels will be implemented; 1D/2D GPU FFT implementation is available though CUDA or ROCm
- multithreading will be explicitly handled, taking into account thread-safety
- host code decomposes and load-balances the G-vectors in “sticks“ of different length between the ranks of a given MPI communicator

- a single G-vector stick is never split between MPI ranks
- communicator of the FFT matches the communicator of the G-vector distribution
- FFT "plan" reserves a right to pre-allocate some CPU and GPU memory and keep it for the entire run

We will design the library according to the following principles:

- use handles (opaque data identifiers) to store information about G-vectors, FFT grids, FFT instances, etc.
- all functions return error codes
- input / output parameters are passed as function arguments
- library should not have a global state
- no exceptions or abnormal terminations

The following minimalistic API is proposed with the idea to create a working "proof of concept" as soon as possible and evaluate its performance and flexibility. In the following months the API will be finalized. SIRIUS DSSDP will be the first to switch to the new FFT implementation and to get rid of the internal FFT3D class.

ft_create_space(*dims*, *mpi_comm*, *execution_device*, *input_location*, *output_location*, *handle*)
Description
Create a handle for the FFT work space. The FFT space handle is used to store the work buffers for the CPU and GPU FFT executors for the maximum grid dimensions provided by *dims*.
Parameters:

dims	[in]	maximum FFT grid dimensions
mpi_comm	[in]	MPI communicator for the parallel transformation and G-vector distribution
execution_device	[in]	type of execution device: CPU or GPU
input_location	[in]	expected location of the input data: CPU, GPU or both
output_location	[in]	expected location of the output data: CPU, GPU or both
handle	[out]	FFT work space handle

gv_create(*mpi_comm*, *dims*, *ngv*, *gv*, *reduce*, *handle*)
Description
Create a lightweight handle for the existing set of **G**-vectors that describe the reciprocal Fourier components of the functions being transformed and bind this **G**-vector set to a particular FFT grid dimensions. The set of **G**-vectors is generated by the host code. It is assumed that the **G**-vectors are distributed between the MPI ranks of the underlying FFT grid communicator.
Parameters:

mpi_comm	[in]	MPI communicator for the parallel transformation and G-vector distribution
dims	[in]	actual dimensions of the FFT grid used for the transformation
ngv	[in]	local number of G-vector for this MPI ranks
gv	[in]	G-vector Miller indices stored as a (3, ngv) integer array
reduce	[in]	indicates if G-vectors are reduced by inversion symmetry or not; if true, the G and -G vectors are treated simultaneously and only G-vectors are taken.
handle	[out]	handle of the G-vector set

ft_create_executor(*fft_space_handle*, *gv_handle*, *handle*)
Description
Create a lightweight handle for the FFT executor. The executor will store the FFT plans for CPU and GPU transformations for the specified G-vector distribution. The work buffers will be taken from the *fft_space_handle*.
Parameters:

fft_space_handle	[in]	handle of the FFT work space
gv_handle	[in]	handle of the G-vector set
handle	[out]	handle of the FFT executor

ft_execute(*handle*, *dir data_g*, *data_r*)
Description
Execute a forward or backward Fourier transform.
Parameters:

handle	[in]	handle of the FFT executor
dir	[in]	direction of the transformation: +1 for $\exp(+i\mathbf{Gr})$ – inverse transform, -1 for $\exp(-i\mathbf{Gr})$ – forward transform
data_g	[inout]	plane-wave expansion coefficients of the function
data_r	[inout]	values of the transformed function on the real-space regular mesh

The following pseudo code shows the usage of the proposed API:

```
! pick a grid size
dims = (100, 100, 100)
! create a work space
ierr = ft_create_space(dims, MPI_COMM_WORLD, "cpu", "cpu", "cpu", work_handle)
! create handle for the not reduced G-vector set
ierr = gv_create(MPI_COMM_WORLD, dims, ngv, gv, .false., gv_handle)
! create FFT executor (FFT plan in the FFTW terminology)
ierr = ft_create_executor(work_handle, gv_handle, fft_exec)
! fill the buffer with plane-wave coefficients
f_in_pw(1:ng)=random()
! execute the G -> r transformation
ierr = ft_execute(fft_exec, 1, f_in_pw, f_r)
! execute the r -> G transformation to a different output buffer
ierr = ft_execute(fft_exec, -1, f_out_pw, f_r)
! compare the results
diff = sum(abs(f_in_pw(1:ng) - f_out_pw(1:ng)))
! check the difference
if (diff > 1e-10) then
  print("Failure")
else
  print("OK")
endif
```

We present here as an example the API of the quantum engine of `BigDFT`.

```
type(dictionary), pointer :: options
type(run_objects) :: run_obj !< the two runs parameters
type(state_properties) :: outs
![...] constructors
call bigdft_init(options)
call run_objects_init(run_obj,options)
call init_state_properties(outs, natoms=bigdft_nat(run_obj))
![...]
! run of the quantum engine
call bigdft_state(run_obj,outs,istat)
!accessors (examples)
!define pointers towards the atomic positions
rxyz_ptr => bigdft_get_rxyz_ptr(run_obj)
!deepcopy of the positions in array pos
call bigdft_get_rxyz(run_obj,rxyz=pos)

!setters (examples)
!fill the atomic positions after they have been modified
call bigdft_set_rxyz(run_obj,rxyz=pos)

!destructors
call free_run_objects(run_obj)
call deallocate_state_properties(outs)
call bigdft_finalize(ierr)
```

5.2 API for parallel linear algebra

Linear algebra is of course one of the fields in which many HPC solutions have been established and highly optimized computational kernels are available on all relevant hardware architectures. We therefore do not aim at contributing to the field by trying to provide different solvers of by implementing new algorithms. However, we observe that the very fact that so many solutions already exist also introduces a significant burden on our community by introducing the need to interfacing to these different solutions and to keep track of their evolution. While this might be required for some of the more basic linear-algebra operations like matrix-multiplications in order to ensure a best possible match of data-structures and to harvest the performance required, the situation is somewhat different for more complex operations. Here we have already identified the problem of solving an eigenvalue problem for a dense matrix. This problem is for example solved in FLEUR as well as in QUANTUM ESPRESSO and both codes provide libraries that construct wrappers around the underlying computational kernels provided by highly specialized math-libraries like LAPACK, ELPA, MAGMA, SCALAPACK and others. We will consolidate these wrappers and construct a common API reflecting the clear structure the underlying mathematical problem to ease the burden to adjust the quantum engines to these different low-level libraries.

While the exact definition of the API will not impose a significant challenge due to the clearness of the underlying mathematical problem, we decided to postpone this step until the underlying data-structures have been consolidated across codes interested in this effort and the challenges due to the performance portability issues addressed in WP2 in the context of linear algebra are clearly identified. As a first estimate we expect to provide at least simple interfaces in which the matrices can be provided distributed in a block-cyclic manner as required by SCALAPACK. Additional interfaces to deal with matrices stored in device memory or in other distributions will then be added as needed in the course of the development.

5.3 API for Poisson Solvers

We here describe, as a futher example, the API of the `PSolver` library that has emerged as a module of the BigDFT code.

The basic quantities that drive the usage of the solver are stored in the opaque object (Fortran datatype) `coulomb_operator`. This object is "opaque" in the sense that the user is not supposed to set directly the components, but via routines of the `Poisson_Solver` Fortran module. This object has to be initalized to define the particular system in which the array containing the charge density is defined. Such initialization is separated in two steps. The first step is associated to the `pkernel_init` function, which sets the internal input parameters of the opaque datatype. Then the `pkernel_set` routine has to be called to allocate the required internal arrays needed to perform the operations. Such scheduling of the initialization enables one to separate the reading of the input parameters from the actual memory storage of the internal arrays. The routine `pkernel_free` is then responsible for freeing such memory storage.

Here follows an example of a solution of a Poisson Equation in vacuum given an array with a density on a uniform grid.

```
call dict_init(inputs) !default values (inputs={})
!override if willing (example of GPU)
 if (gpu) &!in python it would be inputs['setup']['accel']='CUDA'
       call dict_set(inputs//'setup'//'accel', 'CUDA')
kernel=pkernel_init(mpirank,mpisize,inputs, & !setup
                    geocode,ndims,hgrids, & !geometry
                    alpha_bc=alpha,beta_ac=beta,gamma_ab=gamma) !optional
!free input variables if not needed anymore
call dict_free(inputs)
![...] do other stuff here
call pkernel_set(pkernel,verbose=.true.) !allocate buffers (verbosely if you
    like)

![...] from this point you need to allocate (and fill rhoV array as you like)
!transform density in potential
call Electrostatic_Solver(pkernel,rhoV,energies)

!this is like print (little advertisement of yaml emitter in FUTILE)
call yaml_map('The hartree energy of this run is',energies%hartree)

![...] end of usage of the solver
call pkernel_free(pkernel) !release buffers
```

5.4 General common API for the quantum engines

The possibility to provide access to instantiate quantum engines inside third party codes and access to their internal functionalities is one of the strategies that we aim at implementing to provide exascale technology to other developers.

The API for such use should be very general and allow for the use of different quantum engines. The external data types provided in input and output should allow the calling application to store together with the general data all those information that are specific to a given quantum engine or a given hardware of software architecture, but for what concerns all specific data the data type must be opaque, the API should thus provide handles to manage the specific data.

On general grounds the API for the quantum engine will provide initialization, computation, and extra data extraction routines. Schematically, for the common case in which

the quantum engine acts as a forces and stress calculator, and optionally can generate other useful information such as charge density or a density of states (DOS):

```
call init_engine( comm, {state_handle} )
call get_forces_and_stress( comm, structure, {state_handle} ; fa, stress )
call get_charge_density( comm, {state_handle}, charge_density )
call get_dos ( comm, {state_handle}, dos )
```

In the above toy example:

- Variables dealing with forces, stress, charge density, and DOS are considered to have a common structure for all codes, as they refer to universal concepts in the electronic-structure domain.
- `comm`: a structured data-type used to pass the parallelism context and settings to the quantum engine or retrieve it as output; the organization of the parallelism may be described by a general parent communicator plus an application specific descriptor that is hidden and accessible only by specific interfaces.
- `structure`: a structured data-type containing all information to be passed as input argument containing the description of the atomic structure of the simulated system; the descriptor shall be general enough to be compatible with all flagship codes, hidden data may be in this case: pseudopotentials, localized basis sets *etc*
- `state_handle`: a structured data-type used to contain implicitly, and opaquely to the client code, the status of the calculation. All calls should use it, as its contents will be appropriately updated by the quantum engine. Initialization data, completed computation steps, pointers to possibly useful results, etc, are kept in this data structure, which is specific for each quantum engine. Issues of persistence, checkpointing, etc, are non trivial in an exascale context and should be attended to with care.

6 Inter-code work-groups

The modules extracted from specific codes will be developed in feedback with the collaborative development actions planned in WP1. These actions will involve collectively at all levels the developers participating in WP1-4. One of the main goals of these collective actions is to organize and evolve the WP1 software platform towards an effective interoperabilty. This will be done implementing common interfaces and testing them with mini-apps and important demonstrative test cases.

The workgroups on FFT and Parallel Linear algebra have the aim to realize and monitor the optimal portability of the most compute intensive functionalities distributed by WP1.

The Work Group on Quantum-engine interfaces will take care to define, implement and test the quantum engine API integration. This work group will also work on the development of reusable libraries for the evolution of atomic structures as a function of total energies, forces and stress, which is an obvious use case for the Quantum Engine interfaces. As the activity of this work group is quite extended we dedicate below a small subsection to outline in more detail our planned activities on this side.

The work group on Symmetries and K-Point will prepare a set of common libraries for the use of symmetries within the codes.

The work group on documentation will provide common formats for documentation of the delivered APIs and take charge of their documentation of the MAX gitLab repository.

6.1 Quantum Engine Interfaces

An in-code external "geometry" loop is found in most codes: the core electronic structure section (quantum engine) is used to get energies, forces, and stresses, to update the coordinates. This ionic "outer loop" of quantum engine operation is the most amenable to be treated by work-distribution ideas, and might be a very practical use-case of exascale machines in the computational materials science domain. A *serial* form of this extra loop can be used for MD and geometry relaxations. Other ionic problems lend themselves to parallel (and thus more scalable) operation: NEB calculations, phonons in the "finite differences" and linear-response modes, calculation of free energies, etc.

More generally, the abstraction through a simple API of the core quantum engine operation enriches the module/component ecosystem to be defined by WP1, which is the target for the "interoperability platform" of Task T1.2. This interoperability might extend to other properties beyond those related to ionic movement.

The initial goals of the Working Group are to define strategies for providing APIs to exploit (initially) the "force/stress calculator" capabilities of quantum engines and to showcase the functionality through the design of "mini-apps" that exercise those APIs.

7 Conclusions

The present software development plan (SDP) mainly describes the foreseen, planned, and designed modularization of the MAX flagship codes targeting the extraction of many important functionalities that will be refactored, maintained, and distributed as autonomous libraries. This is clearly crucial to allow for a sustainable porting of the flagship codes towards exascale HPC systems and for achieving, in the long term, the goal of having high-level electronic structure codes free from architecture specific instructions.

In the short term perspective, WP1 has to address the work needed to keep the codes up to date with the current HPC technologies and able to efficiently exploit them; this is done in collaboration with WP2 (performance portability), WP4 (codesign), and WP6 (scientific demonstration). Concerning the current HPC systems, the emerging accelerator-based heterogeneous architectures have been rapidly adopted in many computing centres worldwide, notably including the largest HPC machines in Europe (pre-exascale systems) and US. This has imposed to all code consortia collaborating in MAX WP1, 2, and 3, and to the HPC experts of WP4 to rapidly adapt MAX codes to NVIDIA-GPU machines, today's most popular and spread heterogeneous architecture, while also getting ready for AMD and INTEL GPUs, at least.

In preparing such GPU-ready version of the code we have tried to avoid at the most the usage of architecture specific solutions, preserving the code portability particularly for high-level quantum-engines. Concerning codes featuring a localized basis set, such as SIESTA, BIGDFT and CP2K, this goal has been addressed and achieved by leveraging

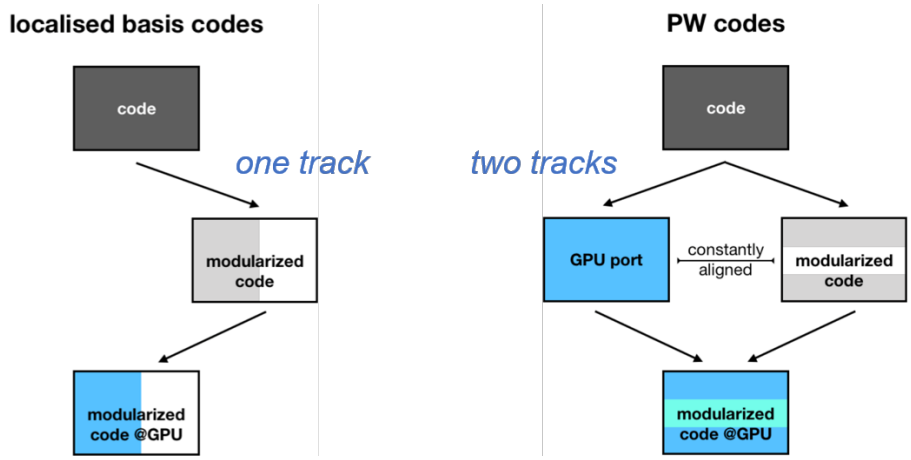

Figure 3: Two routes for refactoring. **Localized Basis** set codes have an access to GPU allocated data only within specific libraries. **Plane Waves** basis codes need to access GPU allocated data through all the code and need a very sparse use of GPU specific constructs.

the encapsulation of mathematical kernels implemented so far, limiting the accelerated code part to the GPU-ready mathematical libraries that are accessed via general architecture agnostic interfaces. Instead, for codes based on plane wave basis sets (QUANTUM ESPRESSO, YAMBO, and FLEUR) – in order to avoid inefficient data movement between host and device memory – it has been necessary to operate on the GPU-allocated data also at higher levels of the code and to resort to well-established architecture-specific programming models (e.g. CUDA and CUDA-Fortran, provided by the PGI-NVidia compiler).

This scenario poses concerns about the code portability for the other emerging architectures. On this side we have undertaken two basic actions:

- for what concerns WP1 libraries, the most common offloading constructs of YAMBO and QUANTUM ESPRESSO have been collected in a common library providing them with architecture agnostic APIs (`DeviceXlib`);
- The usage of more portable and open programming models such as OpenMP (4.5/5) or OpenACC, in particular for the acceleration of loops via preprocessor directives, has been investigated.

In particular, experimentation with OpenMP (4.5 or 5) or OpenACC is already ongoing, but it is necessary that their implementation in available Fortran compilers becomes more stable before we can confidently use it for productions codes.

One last important point regarding the plan presented above is the necessity to monitor and assess the progress of WP1 in completing its tasks. In the first part of WP1 operation we will use the number of functionalities effectively covered by libraries, comparing it with the timeline presented in table 1. In the second part, when the activities will be more dedicated to the improvement of the APIs, the focus will be on performance portability and reusability of the libraries outside their original scope – comparing it with what prospected in fig. 2.

Acronyms

API Application Programming Interface. 4–10, 18, 21, 23, 26–29

CECAM Centre Européen de Calcul Automatique et Moléculaire. 7

DFPT Density Functional Perturbation Theory. 17, 18

DSSDP domain-specific software development platform. 5, 21, 24

ELSI ELectonic Structure Infrastructure [2]. 18

HDF5 Hierchical Data Format v. 5, https://www.hdfgroup.org/solutions/hdf5/. 17

HPC High Performance Computing. 4, 29

HTC High Throughput Computing. 21

RCI Reverse Communication Interface. 18

SDP Software Development Plan. 3, 29

TDDFPT Time Dependent Density Functional Perturbation Theory [4]. 17

XML Extensible Markup Language. 17

References

[1] ESL. URL https://esl.cecam.org/Main_Page.

[2] ELSI. URL http://elsi-interchange.org.

[3] Videau, B. *et al.* Boast: A metaprogramming framework to produce portable and efficient computing kernels for hpc applications. *Int. J. High Perform. Comput. Appl.* **32**, 28–44 (2018).

[4] Rocca, D., Gebauer, R., Saad, Y. & Baroni, S. Turbo charging time-dependent density-functional theory with Lanczos chains. *J. Chem. Phys.* **128**, 154105 (2008).