D1.3

Second report on software architecture and implementation planning

Stefano Baroni, Uliana Alekseeva, Augustin Degomme, Pietro Delugas, Stefano de Gironcoli, Andrea Ferretti, Alberto Garcia, Luigi Genovese, Paolo Giannozzi, Anton Kozhevnikov, Ivan Marri, Davide Sangalli, and Daniel Wortmann

Due date of deliverable	31/5/2020 (**month 18**)
Actual submission date	31/5/2020
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.	D1.3

Authors	Stefano Baroni, Uliana Alekseeva, Augustin Degomme, Pietro Delugas, Stefano de Gironcoli, Andrea Ferretti, Alberto Garcia, Luigi Genovese, Paolo Giannozzi, Anton Kozhevnikov, Ivan Marri, Davide Sangalli, and Daniel Wortmann
To be cited as	Baroni et al. (2020): Second report on software architecture and implementation planning. Deliverable D1.3 of the H2020 CoE MaX (final version as of 31/05/2020). EC grant agreement no: 824143, SISSA, Trieste, Italy.

Disclaimer

Contents

Executive Summary

This document updates the SDP [1] of the MAX flagship codes and libraries released at month 6 (May 2019) and reports on the progress of the development of the MAX libraries. The activities are generally on schedule with the plan presented in the SDP. An update on the development of the libraries is presented in detail in section 2.

The work done on libraries up to this point provides us with a large number of actual cases that have been useful to clarify and update the concepts and practices followed in our development program. The main lessons learned are discussed in section 3. The points discussed in this section are:

- **Libraries and Modules.** Together with the *libraries* completely encapsulated and exchanging data only through the APIs, other code portions, the *modules*, have been partially encapsulated but also access the global data structure of the code. Libraries are more easily used in other codes while the modules strategy may be more performing.
- **Descriptors, structures, and data passing.** Opaque handles are used when one needs to enhance data encapsulation; Fortran data structures are used instead when it is preferable to keep visible to programmers the content of descriptors and data structures.
- **Usage of dictionaries for more flexible APIs.** Leveranging and extending the implementation in the `Futile` library we are experimenting the usage dictionaries for the realization of APIs with more flexibility in the types and number of arguments.
- **How to export interfaces.** Most of the libraries keep the original interface exposition via Fortran Modules, when it is important to avoid toolchain dependencies the interfaces are exported using C style header files.
- **Which build system to use** `Autoconf` and `CMake` build systems are used in most of our libraries. While the first one is more intuitive, `CMake` is more suited for modular build systems.
- **Usage of external libraries and collaboration with other projects** The refactoring has allowed to enhance performance also using libraries developed outside the MAX project. In this context is important to promote the interaction with external projects such as ESL and others.

1 Introduction

This document is an update of the Software Development Plan [1] (SDP), delivered at month 6 (May 2019). We also report here the progress made in the plan implementation within the first 18 months.

The SDP released at month M6 (May 2019) of the project designed an extended refactoring of the MAX flagship codes, aimed at modularize functionalities, encapsulate data structures and adopt general reusable APIs to access the functionalities and operate on the data structures.

Such refactoring and modularization involved mainly the mathematical compute intensive parts of the code and an utility layer that could allow the programmers to access in an uniform and flexible way a set of basic functionalities that are needed ubiquitously through all the code. This utility layer includes constructs for accessing MPI and OpenMP parallelism, error handling, I/O operations. To cope with heterogeneous architectures based on accelerators, the utility layer is now being extended with new APIs for masking the most recurrent constructs used to move, operate, and synchronize data in multiple address spaces (*see* section 2.2.4).

In order to make such work on refactoring useful to a broader community the SDP identified within the modularized functionalities those most suited for being distributed as autonomous libraries and tools and whose development and release could proceed alongside with the refactoring of the flagship codes. The SDP defined a set of criteria to be fulfilled by the libraries and the APIs, mostly based on the concepts of autonomy and efficiency. Depending on the advancement of the development the libraries are classified as *proof of concept*, *Beta stage*, and *production* libraries.

We have organized this report in three sections. In section 2 we will report in detail the status of our libraries. Then in section 3 we discuss the main issues, solutions, practices and general concepts that we have learned during these 18 months of development. The concluding section 4 will summarize the plans for the next 18 months focusing on what needs to be updated or added with respect to the first SDP.

2 Libraries

The MAX libraries provide a wide set of functionalities frequently used in electronic structure simulation codes. These libraries, extracted from the flagship codes of the MAX centre, are constantly updated and maintained by the developers of these codes and aim to make the efforts of the center directly available to the developers of electronic structure codes.

In many cases these libraries are derived directly from the extraction of the functionalities already present in the flagship codes; this has caused redundancy in the coverage of some of the basic functionalities. During the first part of the project we have chosen to not unify the overlapping libraries because this could have represented a major complication for the refactoring action.

Entering in the second stint of the project we have started to put more effort for unifying these functionalities adopting common interfaces and when possible unifying the packages in unique libraries. Notable example of this effort is the work done for the LAXlib library (*see* section 2.1.1) for the unification of the linear algebra functionalities

extracted from FLEUR and QUANTUM ESPRESSO.

This section presents in detail an update of the status of the various packages developed in the first half of the projects. The development schedule of the libraries for the first 18 month was presented in the SDP delivered at M6 in the D1.1 report [1], table 1 summarizes the current status and the planned development of the libraries in the next 18 months.

2.1 Available production libraries

2.1.1 LAXlib

Linear algebra operations are among the most compute-intensive parts of almost all MAX flagship codes. In particular, the solution of a (generalized) Hermitian eigenvalue problem constitutes a significant challenge in many cases. As this is a well-defined and common mathematical problem, a large variety of libraries is available for its solution: from the well-established BLAS and LAPACK library (and their parallel counterpart SCALAPACK) to vendor optimised implementations like Intel-MKL, AMD AOCL, IBM ESSL, NVIDIA cublas and cuSolver, and more modern (and domain specific) approaches like e.g. the ELPA library or the Elemental library. This zoo of solutions is partly due to the fact that these libraries target different computational architectures and hence optimal performance cannot be archived by using the same solver in all cases. Therefore, the actual computational kernel for this problem is not our focus of attention but rather the question of how to interface to these libraries efficiently and how to make these different solutions easily available for our codes and for the community in general. This is the main goal of the implementation of the LAXlib component in MAX .

We defined a rather simple API which exposes the different available solvers on a unified high level interface such that our codes can easily exploit the library best suited for the used architecture and problem size. For simplicity, we adopted two different data-layouts in which the matrices can be provided. One set of routines that require the full matrices to be provided and thus is suitable for small problems or shared memory setups and routines that deal with distributed matrices stored across nodes in which we adopted the SCALAPACK distribution scheme with the corresponding BLACS descriptors.

In addition to the computational routines, LAXlib also contains callback routines in order to register code-specific timing as well as routines and error-handlers to allow for a seamless integration into the code-specific infrastructure. The library is constructed without explicit Fortran modules but with general include (header) files such that it imposes no compiler specific dependencies in its usage.

The library is currently interfaced to the QE and the FLEUR codes. Possible adoption is also foreseen for the Yambo code, e.g. to exploit the interface to ELPA. The solver routines available in FLEUR and part of the effort to create a FLEUR-LA library have also been integrated into the LAXlib such that the creation of an additional FLEUR-specific library was no longer necessary and the development of the FLEUR-LA has been abandoned in favour of the joint development reported here.

While the library is used in production already, work to implement interfaces for further solvers is ongoing and some additional modifications for the efficient use of GPU offloading and other compute concepts with non-trivial memory hierarchy are explored and will be included in future versions of the library.

MAX libraries expected roadmap (M18-M36)

Library	Group	Month M18	Month M24	Month M36
FUTILE	BIGDFT	Production	Production	Production
PSolver	BIGDFT	Production	Production	Production
atlab	BIGDFT	PoC	PoC	Beta
libconv	BIGDFT	Production	Production	Production
bundler	BIGDFT	Beta	Production	Production
PyBigDFT	BIGDFT	Beta	Production	Production
sphinx-fortran	BIGDFT	Beta	Beta	Production
juDFT	FLEUR	Production	Production	Production
LAPWlib	FLEUR	Beta	Beta	Production
IO-t	FLEUR	Beta	Beta	Production
qe_h5	Q. ESPRESSO	Production	Production	Production
xmltool	Q. ESPRESSO	Production	Production	Production
UtilXlib	Q. ESPRESSO	Production	Production	Production
FFTXlib	Q. ESPRESSO	Beta	Beta	Production
LaXlib	Q. ESPRESSO	Production	Production	Production
KS_solvers	Q. ESPRESSO	Beta	Production	Production
LRlib	Q. ESPRESSO	PoC	PoC	Beta
UPF_lib	Q. ESPRESSO YAMBO	PoC	Beta	Production
XCfunc_Xlib	Q. ESPRESSO YAMBO	PoC	Beta	Production
DevXlib	Q. ESPRESSO YAMBO	Beta	Beta	Production
Driver_Ylib	YAMBO	PoC	Beta	Production
CoulCut_Ylib	YAMBO	PoC	Beta	Production
LA_Ylib	YAMBO	PoC	Beta	Production
IO_Ylib	YAMBO	PoC	Beta	Production
GridXC	SIESTA	Production	Production	Production
libPSML	SIESTA	Production	Production	Production
ELSI-interface	SIESTA	PoC	PoC	PoC
LibNeigh	SIESTA	PoC	Beta	Production
Lua scripting	SIESTA	Production	Production	Production
libFDF	SIESTA	Production	Production	Production
xmlf90	SIESTA	Production	Production	Production
libDBCSR	CP2K	Production	Production	Production

Table 1: Expected Roadmap for the libraries development in the second 18 months of the project. **PoC :** *Proof of concept* version, **BETA:** release candidate, **Production:** interoperable library ready for release.

2.1.2 SpFFT

SpFFT is a 3D FFT library for sparse frequency domain data written in C++ with support for MPI, OpenMP, CUDA, and ROCm. It was originally intended for transforms of data with spherical cutoff in frequency domain, as required by some computational materials science codes. For distributed computations, SpFFT uses a slab decomposition in space domain and pencil decomposition in frequency domain (all sparse data within a pencil must be on one rank).

To allow for pre-allocation and reuse of memory, the design of the library is based on two classes:

- *Grid*: Allocates memory for transforms up to a given size in each dimension.
- *Transform*: Is associated with a *Grid* and can have any size up to the *Grid* maximum dimensions. A *Transform* holds a counted reference to the underlying *Grid*. Therefore, *Transforms* created with the same *Grid* share memory, which is only freed once the *Grid* and all associated *Transforms* are destroyed.

The user provides memory for storing sparse frequency domain data, while a *Transform* provides memory for space domain data. This implies that executing a *Transform* will override the space domain data of all other *Transforms* associated with the same *Grid*. As a proof-of-concept SpFFT library was interfaced with FLEUR, Yambo, and Quantum ESPRESSO. The SpFFT library is used in production in SIRIUS.

The API documentation is available here:
https://spfft.readthedocs.io/en/latest/?badge=latest

The library development is hosted here:
https://github.com/eth-cscs/SpFFT

2.1.3 COSMA

COSMA is a parallel, high-performance, GPU-accelerated, matrix-matrix multiplication algorithm that is communication-optimal for all combinations of matrix dimensions, number of processors and memory sizes, without the need for any parameter tuning. The key idea behind COSMA is to first derive a tight optimal sequential schedule and only then parallelise it, preserving I/O optimality between processes. This stands in contrast with the 2D and 3D algorithms, which fix process domain decomposition upfront and then map it to the matrix dimensions, which may result in asymptotically more communication. The final design of COSMA facilitates the overlap of computation and communication, ensuring speedups and applicability of modern mechanisms such as RDMA. COSMA permits to not utilise some processors in order to optimise the processor grid, which reduces the communication volume even further and increases the computation volume per processor. COSMA is written in C++ with support for MPI, OpenMP, CUDA, and ROCm.

In the past months the work to fully integrate the COSMA library in CP2K code has been accomplished. Dependency on the COSMA library and *pdgemm* wrapper was added to CP2K build tool-chain, all unit tests were verified and performance in production cRPA runs of 128 water molecules has been measured.

The library and documentation are hosted here:
https://github.com/eth-cscs/COSMA

2.1.4 DBCSR

DBCSR is a library designed to efficiently perform sparse matrix-matrix multiplication, among other operations. It is MPI and OpenMP parallel and can exploit NVIDIA and AMD GPUs via CUDA and HIP. DBCSR is used in production by CP2K code but can also be interfaced with SIESTA and other codes that use localised basis set. In the past months the work on auto-tuning of compute intensive GPU kernels of small matrix-matrix multiplications has been accomplished and a ROCm port was added as library's back-end. A lot of documentation was written for the auto-tuning and predictive modelling frameworks.

Documentation available here:
https://cp2k.github.io/dbcsr/develop/index.html).

The DBCSR library is hosted here:
https://github.com/cp2k/dbcsr

2.1.5 xmltool

This is a general purpose python tool that automatically generates the modules needed for XML I/O for generic codes. The tool may be used by any Fortran code and depends only on the given XSD schema; this latter is the only input needed and must be provided using the XSD schema [2] language specifications.

This tool is released in production phase. Templates are currently provided only for the FoX library, but they are in part reusable also with the xmlf90 library provided in MAX components. Work is ongoing to leverage the `xmlschema` package to improve maintananility and flexibilty. Further work on new templates in ongoing in order to optimize the usage with `xmlf90` and to implement the usage of C libraries like `libxml` and `gdome`

The templace of the produced Fortran code is provided by a set of `Jinja` files. The current templates generate 5 modules:

- `types_module`: contains the Fortran types, i.e. the type structure mirroring the content of the XSD types as described by the schema provided in input.
- `read_module`: provides the `read` interface for the FoX implementation of the DOM API. The interface accepts as argument a pointer to a DOM node, a structured data type corresponding to the element one wants to read, and an optional error flag. For the reading to be successful the DOM node must point to an element of the type corresponding to the Fortran type of the variable given as argument; if the error flag is present, the flag reports the success (0) or the failure of the operation ($\neq 0$), otherwise the failure causes a stop of the execution.
- `write_module`: provides the `write` interface containing the FoX instructions to write the given XML element; the interface accepts as input a FoX descriptor to

an open XML file and a Fortran structure containing the data to be written, together with an optional error flag.

- `init_module`: provides the `init` interface. This interface is used to initialize a Fortran data type corresponding to the XML elements; apart from the object which one has to initialize, the other arguments depend on the content of the type.
- `bcast_module`: provides an interface to broadcast the content of a data type element by element; the related MPI communicator is provided as argument to the routine.

2.1.6 qe_h5

This small library provides a minimal, easy to use, API to write, read, and modify HDF5 files. The APIs support read and write access to groups and datasets, and to their attributes. It covers the use of all the main Fortran datatypes. The support for the COMPLEX type is implemented just doubling the dimension in the first direction of the datasets and reading/writing them as real. `COMPLEX` and `REAL` types are currently assumed to be double precision. Strings are written as fixed length but the library is able to read also variable-length strings, which guarantees compatibility with files written by other codes.

The API provides descriptors for handling files, groups, and datasets. Passing these descriptors, the interfaces allow for: opening and closing files and dataset; writing and reading attributes; writing and reading datasets; reading and writing hyperslabs.

A description of the API may be found on the MAX gitlab repository [1]

Ongoing work: A few improvements to the library are under way. We want to support the following HDF5 features:

- single-precision datatypes;
- compressed datasets;
- parallel I/O.

We are also working to avoid the usage of the HDF5 Fortran modules, binding directly the C routines in order to make linking to the precompiled library easier.

2.1.7 juDFT library

The juDFT-library has been separated from the FLEUR code, can be used and compiled independently. It provides standard functionality used by FLEUR and other juDFT codes for error-handling, timing, and the automatic collection of usage data of the code. Also integrated is an easy-to use wrapper layer on top of HDF5 IO to enable the use of this library without the need to use the low-level hdf5 interface. In particular, this wrapper simplifies the handling of IO of array sections with the additional mapping of complex data to the underlying HDF5 datatypes. While this HDF5 interface is widely used, it is

[1] https://gitlab.com/max-centre/components/qeh5_lib/-/wikis/APIs-and-descriptors

an optional component of the library such that the juDFT-library can also be deployed if no HDF5 is available on the system or is not needed.

2.1.8 SIRIUS

SIRIUS is a domain-specific software development platform for electronic structure calculations. It implements pseudopotential plane wave (PP-PW) and full potential linearised augmented plane wave (FP-LAPW) methods and is designed for GPU acceleration of popular community codes such as Exciting, Elk, and Quantum ESPRESSO. SIRIUS is written in C++11 with MPI, OpenMP, and CUDA/ROCm programming models. SIRIUS is organised as a collection of classes that abstract away the different building blocks of the DFT self-consistent cycle.

The SIRIUS-accelerated Quantum ESPRESSO code is used in production at CSCS. SIRIUS is also interfaced with the CP2K code extending its capabilities with full-potential and pseudopotential plane-wave DFT solvers. In the past months the work on numerical reproducibility of native Quantum ESPRESSO has been accomplished and a collection of verification tests to benchmark QE-6.5 and QE-SIRIUS has been created. These tests are hosted here:
https://github.com/electronic-structure/q-e-sirius-test,

SIRIUS is hosted here:
https://github.com/electronic-structure/SIRIUS

and the API documentation can be found here:
https://electronic-structure.github.io/SIRIUS-doc/

2.1.9 libGridXC

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. Updates to the building system and documentation have been carried out since M12. The library is in production and stable. The next major user-facing upgrade will be the implementation of mgga functionality, and a new mechanism for extraction of derivatives of various orders. This is going to need a refinement of the API, along the lines mentioned in section 3 below.

2.1.10 xmlf90

`xmlf90` is a suite of light-weight libraries to read and write XML documents in Fortran. The library is in production, and stable. It is planned to upgrade the internal mechanisms for string handling to remove some remaining hard-coded length limits. This new string technology will be also used to upgrade the DOM subsystem of the library.

2.1.11 libPSML

This library is the main piece of the ecosystem of tools to handle pseudopotentials in the PSML format (see http://esl.cecam.org/PSML). The library is quite stable, with the

API unchanged since its original release. As it makes heavy use of Fortran optional arguments, a direct mapping of the interface to C would require Fortran 2008 plus the TS 29113 extension. This is not yet advisable if we want to keep the library portable to systems with varying degrees of standard compliance in their compilers.

2.1.12 libFDF

This library provides an interface for processing input files in the FDF format used by SIESTA. Stable and in production for a very long time, but only recently extracted as a independently released package. More documentation is being produced.

2.1.13 Lua Scripting Interface

Lua is an easy-to-learn and fast scripting language built for embedding. It is very lightweight (its memory footprint is less than 300kB), and provides very simple ways to interface to the data structures and routines of a host program. A Lua script, interpreted by the Lua interpreter embedded in the program, can then control the flow of execution and the data. Different user-level scripts can implement new functionalities, *without recompilation of the host code*. The **flook library** enables Fortran and an embedded Lua interpreter to communicate in a seamless way by passing variables to and from "tables" in Lua.

The strategy we have followed in SIESTA is based on handling control to the Lua interpreter at specific relevant points in the program flow (e.g. at the beginning of a geometry step, at the end of a scf step, etc). Lua scripts implement handlers appropriate to the point they want to hook into, and can request access to specific data structures. For example, a script intended to implement a better scf mixing algorithm would be executed after every scf step, inspecting the convergence data, and changing mixing parameters or schemes, as appropriate. As another example, convergence checks over mesh-cutoffs and k-point sampling can be performed automatically. The above mixing scenario exemplifies an important area of usefulness of the approach: the prototyping in Lua, (followed eventually by a full implementation), of new ideas and algorithms.

We have implemented a number of custom molecular dynamics modes, geometry relaxation algorithms, and advanced optimization schemes, in a pure Lua library (FLOS). The code in the library can be re-used, or taken as starting point for other implementations by users. These user-level scripts can in turn be shared, opening the way to the development of new functionality with faster turnaround that the traditional approach that needs a careful integration into the program's code base. As a specific showcase of the power of the Lua embedding, we have developed a number of variations of the nudged-elastic band method (NEB) for transition-state search. Previously proposed implementations in SIESTA involved significant, hard to maintain code changes, and did not make it into the mainstream version. With Lua, we have been able to implement, non-intrusively, not only the standard algorithm, but a Double Nudged Elastic Band (DNEB) variation, and also another version which treats atomic coordinates and lattice variables on an equal footing (the variable-cell NEB, or VC-NEB, method).

2.2 Libraries at beta stage

2.2.1 FFTXlib

This is a high-performance specific library for performing FFT 3D operations than can exploit different kinds of parallelism: pure MPI, hybrid MPI+OpenMP, hybrid MPI + CUDA-GPU. It is currently at the beta stage. Work is ongoing for expanding the portability of the library; implementing mixed-precision support; implementing an API complying with the specifications designed in the SDP.

The APIs are based on a general descriptor type which carries the information on FFT mesh dimensions and its distribution over MPI tasks. The API provides then interfaces to initialise the grid, query the presence of sticks or planes inside the MPI node and perform FFT operations.

A full description of the APIs and the descriptors may be found on the MAX repository on GitLab [2]

2.2.2 KS_solvers

This is a collection of iterative diagonalization algorithms to solve the Kohn-Sham equations. The iterative diagonalization algorithms collected inside the `KS_Solvers` are disentangled from the specific Hamiltonian builder, which is called by the library as an external routine; the definition of wavefunctions and their scalar products inside the Hamiltonian builder must be compatible with the one used inside `KS_Solvers`. For some of the algorithms, a Reverse Communication Interface (RCI) is also available, allowing one to directly pass the $H|\psi\rangle$ vectors to the library, leaving to the programmer the task of computing and converting them to the format expected by the RCI.

The possibility of decoupling the `KS_Solvers`, `LAXlib`, and `FFTXlib` libraries from their native codes was first demonstrated during a Workshop organized in 2017 within the ESL initiative [3]. Moreover, both `KS_Solvers` as well as `LAXlib` may use another library maintained by ESL, `ELPA` (included in `ELSI`) for dense-matrix diagonalization.

2.2.3 LAPWlib

The refactoring of complex operations on LAPW wavefunctions that will lead to the creation of the LAPWlib is currently work in progress. Fundamental operations such as the mapping between the muffin-tin representation of the basis functions and the interstitial plane waves are already encapsulated and will be performance tuned for multiple architectures. In a similar state are routines for many of the complex eigenvector operations needed for the hybrid functionals. For these operations a mix-product basis datatype was introduced clarifying the needed datastructures. Most of the remaining work will be dedicated to the finalization of the API, the separation of the code into a library and its tuning and porting.

2.2.4 DevXlib

Performance portability across current and future heterogeneous architectures is one of the grand challenges in the design of HPC applications. General-purpose frameworks

[2] https://gitlab.com/max-centre/components/fftxlib/-/wikis/APIs-and-descriptors

have been proposed [4, 5, 6, 7], but none of them has reached maturity and widespread adoption. In addition, Fortran support is still very limited or missing entirely. In this context, the MAX CoE is promoting and coordinating a collective effort involving the developers of various materials modeling applications. Taking on this challenge with a domain-specific approach has the advantage of providing abstraction and encapsulation of a limited number of functionalities that constitute the building blocks of the most common operations performed on the accelerators in this field. This will allow us to prepare low-level architecture-specific implementations of a limited number of kernels that have been already characterized and isolated, thus keeping the source code of the various scientific applications untouched and reducing code branches when new systems will appear on the market.

Such an effort is still in the early stages, but is under active development and is progressively entering the GPU port of QUANTUM ESPRESSO through the so-called `DevXlib` library. This library started off as a common initiative shared among MAX codes (notably QUANTUM ESPRESSO and `Yambo`), aimed at hiding CUDA Fortran extensions in the main source base. Being used by different codes, the library has been rationalized and further abstracted, thus becoming a performance portability tool aimed at supporting multiple back-ends (support to OpenACC and OpenMP-5 foreseen, direct extension to CUDA C possible). The main features included in the library by design are the following:

- performance portability for Fortran codes;
- deal with multiple hardware and software stacks, programming models and missing standards;
- wrap/encapsulate device specific code;
- focused on limiting code disruption (to foster community support).

It is important to note that part of the library design includes the definition of which device-related abstract concepts need to be exposed to the scientific developers. To give an example, memory copy and synchronization to/from host/device memory are abstract operations that the developers of property calculators or of the quantum engine itself may need to control directly. Therefore, `DevXlib` exposes such control in the form of library APIs that are agnostic of the specific device back-end.

In practice, `DevXlib` provides the user with (*i*) interfaces to *memory handling* operations including creation and locking of memory buffers (`device_memcpy` and `device_buffers`); (*ii*) interfaces to *basic and dense-matrix linear algebra routines*, similarly to BLAS and Lapack (`device_linalg`); (*iii*) interfaces to more *domain-specific operations* (`device_auxfuncs`); (*iv*) device-oriented *data structure* compatible with Fortran usage. In particular, memory handling allows the user to copy memory host-to-host, device-to-device, and also across memories, host-to-device and vice-versa, thereby dealing also with memory off-load and synchronization. Importantly, both synchronous and asynchronous copies can be performed with explicit control. Moreover, the explicit handling of memory buffers is meant to ease or avoid the procedure of allocation and deallocation of auxiliary workspace.

2.3 Proof of concept of oncoming libraries

2.3.1 LR_Modules

This module at the moment kept at a module level. It is a more recent part of QUANTUM ESPRESSO, whose encapsulation started about five years ago and progressed significantly since that time. It intends to provide a unified , harmonic, flexible access to the functionalities that are common to all linear-response and MBPT codes which adopt a data structure compatible with the one adopted in the QUANTUM ESPRESSO suite. `LR_Modules` contains the following functionalities: (i) definition of global data structures for linear response, (ii) calculators of linear-response quantities (such as e.g. response density and potentials), (ii) iterative solvers (e.g. Lanczos recursive algorithms), (iii) response exchange-correlation kernel calculators, (iv) symmetrization routines, (v) projectors on the empty-states manifold, to name a few. The functionalities of `LR_Modules` are used in the following packages:

- `PHonon` for calculation of lattice vibrational modes (phonons), Born effective charges, dielectric tensor, and other vibrational properties;
- `TDDFPT` for calculation of optical absorption spectra of molecular systems, collective excitations in solids such as plasmonsand magnons;
- `EPW` for calculation of electron-phonon coupling, transport, and superconducting properties of materials;
- `HP` for the first-principles calculation of Hubbard parameters of Hubbard-corrected DFT.

The generalised and unified subroutines from `LR_Modules` have been refactored in such a way that they can be easily and straightforwardly employed in any other future linear-response or MBPT code of QUANTUM ESPRESSO or even in third-party codes. They can now be used generically to build perturbations, apply them to the occupied ground-state Kohn-Sham wave functions and compute the related self-consistent first-order response properties either by solving the Sternheimer equations or by solving the Liouville quantum equations using the Lanczos recursion method.

2.3.2 IO-t

Within the set of datatypes of FLEUR a subsection related to input- and setup-types has been identified and the corresponding input has been reshuffled to reflect these data structures. The IO has been encapsulated directly with the data types ensuring consistency of the IO and enabling the simple use of different formats. In a future step we plan to extend this scheme such that flexible storage and communication of the data structures is made possible and to integrate this with a flexible job management mechanism possibly also interfacing a scripting feature.

2.3.3 libNeighb

The neighbor search library, originally a module within Siesta, is in the process of API re-design to decouple it from the use cases in Siesta and make it more generally useful.

2.3.4 ELSI interface

The ELSI interface in Siesta is currently kept at the module level, and its functionality is currently tailored to specific entry points in the program. Making it more abstract would result in a general interface library to ELSI for Siesta-like codes (i.e., codes with sparse Hamiltonians and overlap matrices, stemming from the use of strictly localised orbitals as basis sets). However, the ELSI library itself is providing more and more hooks for sparse operation in its API, so the usefulness of our planned wrapping is diminishing. Further work on the abstraction has stopped.

2.3.5 IO_Ylib

In the process of modularization of the yambo code we extracted the low level subroutines which take care of the I/O. The yambo code takes advantage of NETCDF+HDF5 libraries for the I/O of the binary data. However the explicit calls to the NETCDF API are localised in very few subroutines, namely *io_bulk.F* and *io_elemental.F*, which are then called by other subroutines. These two subroutines have few dependencies and, together with the associated module, *mod_IO.F* have been isolated and we have modified the configure so that the resulting *libio.a* can be compiled independently of all the other subroutines of the code. Starting from this step we plan to work on the subroutines so that they can act as a wrapper to different kinds of I/O schemes without the need of changing the rest of the code. We have identified the possbile options: NETCDF+PNETCDF, NETCDF+HDF5, HDF5, QE_H5+HDF5. In the past the same scheme was used to also support plain fortran I/O, which was later dropped due to the many limitations compared to the present NETCDF+HDF5 scheme.

A similar logical structure has been put in place to accommodate also the other libraries to be extracted from the Yambo code. These libraries are Driver_ylib (handling of the code user command line interface), CoulCut_ylib (handling of Coulomb potential cutoff techniques), la_ylib (linear algebra drivers). Overall, a dedicated git repository is in place and the development of these libraries has been moved there. The goals are two fold: on the one side to have a better handling of the growing complexity of the code (ideally having separate versioning of the libraries and of the core code); on the other side, some of these functionalities can actually be shared with other codes (think eg abut the collection of Coulomb cutoff techniques, which can be easily shared with Quantum ESPRESSO or other plane wave codes).

3 Update on the APIs, factorisation, and interoperability

Libraries and modules. Refactoring of the codes has enhanced the data encapsulation and has defined effective and portable APIs. One notable result of this effort on refactoring are the large number of libraries that have so far reached the production or beta release phase and can already be used in external codes with very small refactoring. The data encapsulation and APIs definition of general APIs has also involved other parts of the codes –which we will refer to as modules– that for computational efficiency maintain access to main data structure of the program. Usage of such modules is in the current stage thus more oriented towards codes having the same data structures, while for for third party codes it would require a major refactoring. For many of the modules further

work is planned in order to reach the full encapsulation of the data structure fulfilling all the authonomy criteria for the libraries.

Descriptors, structures, and data passing. Both for libraries and modules the APIs effectiveness is enhanced adopting descriptors, data structures for passing parameters, settings and all the input and output to and from the library or module. This is obviously crucial for the libraries as all communication passes through the APIs; but it is also important for modules because effective APIs make them flexible, reusable, more intuitive, and less bug prone. Depending on the case, in WP1 we have been adopting as descriptors either *o*paque handles or Fortran data-structures.

Opaque handles are common practice for software engineering, they are used in well established libraries as MPI, Scalapack, and HDF5. Handles are usually integers or integer arrays. The content of the handles can be used directly or as resource handles used to manage records allocated inside the data structures of the libraries and modules. Opaque handles must be initialized with an API call; the whole lifetime of the the structure: creation, modifications and destruction are thus completely managed internally by the library and protected by unchecked modifications.

While in the initial SDP [1] it was provided as general guideline to adopt whenever possible opaque handles instead of Fortran data structures, during the development we have found that in many cases the usage of Fortran data structures allows to maintain the development at a more intuitive level and allows a more immediate debugging. For these reasons, in modules and in beta-stage libraries Fortran data structures are still used, while the adoption of opaque handles will be addressed when the APIs and the structure of the codes will have reached the desired stability.

Interfaces via header files or Fortran modules. There are two main choices on how our libraries export their interface: header files; Fortran *modules*. Most of the libraries provided by the MAX WP1 are written in Fortran; the designed way to export their APIs is thus via the Fortran modules. The modules provide a visible specification of the variables and interfaces that has been imported; they also allow to expose with the desired level of protection the data structure of the libraries, this is still necessary for modules and for libraries not yet at the production stage. Fortran modules have the main drawback to impose a strict toolchain dependence between the library and the programming code, because modules must be compiled with a compiler compatible with the one used to compile the main program.

Header files are instead the standard of C and C++ programming. They have several advantages in term of portability and flexibility but render the import of interfaces less visible. The usage of header files has been preferred in many of the compute-intensive libraries *e*.g. LAXLIB whose performance is crucial and require a careful compilation and tuning. The usage of header files is indeed instrumental for achieving tool-chain independence, although not sufficient, because tool-chain dependencies may be introduced also by I/O operations or other system calls which thus should be avoided or properly initialized in so as they are fully compatible with the rest of the program. It is also possible to use mixed approach writing a small interface module to be used by the hosting code to import the interfaces. The interface module would import the headers from the library and should be compiled together with the hosting code.

Other ideas in API design. One issue that is apparently closer to the scientific content of a library but that affects more fundamental aspects of interfacing is the control of options or required scientific use-cases. A case in point is the PSolver library, which offers a wide range of options for solving the Poisson equation. Some of the options are incompatible with each other, and there are many of them, so a traditional approach ("one argument per option") would make the API unwieldy. More importantly, API changes would be needed with every refinement of the underlying physical model, to add more options. The solution adopted by PSolver is based on a dictionary of options, which is passed as a single argument to the library. The dictionary can be serialized in YAML for humans, and handled by the code with appropriate data structures.

The other data required by PSolver are typically standard arrays holding charge densities, potentials, etc, which are always required and thus a fixture of the API. In other libraries the number of pieces of domain data (typically arrays) needed on input or returned on output depends on the use case. For example, in a library computing the XC energy and potential (e.g. libGridXC), it might be possible to do it with various levels ("rungs") of theory: LDA, GGA, MGGA, etc. Each might require different input (gradient of the density for GGA, gradient and laplacian of the density, and kinetic energy density for MGGA, etc). Also, it might be possible to request various orders of derivatives of the XC objects (again, potentially with respect to density, gradient, laplacian, etc). In the general case one would end up supporting several dozen optional array arguments in the API.

While it is in principle possible to extend the dictionary concept to hold pointers to arrays in the "value" field, this might not be the best solution for the API. Instead, one can consider a more abstract interface based on the concept of "calculation object" which is fed, apart from appropriate physical options (maybe still with a YAML-based dictionary), settings regarding the computation and the associated data. Schematically:

```
...
call init_xc_calc(calc)
call set_calc_level(calc,"GGA")
call set_calc_functional(calc,"PBE")
call set_calc_density(calc,rho(:,:,:))
call set_calc_gradient(calc,grad_rho(:,:,:))
call set_calc_output_vxc(calc,vxc(:,:,:))
...
call run_calc(calc)
...
```

In the above example, options and data links are performed by routines that modify internal data structures in the `calc` object. Those features not explicitly invoked retain their default settings. Hence new features and functionalities can be implemented while maintaining backwards compatibility. In Fortran, one could implement the above scheme in a module holding the `calc` type definition and the associated routines. Modern standards (starting with the almost universally supported Fortran2003) offer the possibility of encapsulating the functionality in an equivalent "object oriented" form, which might have some advantages for code organization and clarity.

```
...
```

```
call calc%init()
call calc%set_level("GGA")
...
```

APIs following this concept have the added advantage of allowing concurrent execution, by eliminating global variables. One could have two instances of XC calculators with different levels of theory, for example.

Build systems Two main build system have been used by our codes and libraries: `Autoconf` and CMake. Autoconf works in a more intuitive manner, it is easier to debug and, when needed, expert users can also modify manually the configuration files produced by the tool. On the other hand the tool is conceived for a monolithic compilation, inserting external submodules is cumbersome.

CMake is a more modern tool, conceived for a modular build system. This facilitates the usage of submodules as well as the incorporation of the libraries as submodules. Its usage and debugging is less intuitive as the build mechanism produces a large number of configuration files that are difficult to edit manually.

4 Conclusions and ongoing work

In this first half of the project the development and release of the libraries has proceeded as scheduled in the SDP and the planned targets have been reached. The criteria for autonomy and interoperability proposed in the SDP have been an useful and affordable guideline for the development.

A significant number of libraries, in production or in Beta stage, are now available for external development and are instrumental to the thorough modularization of the flagship codes.

Many of the libraries which are still at the proof of concept stage are already used inside the flagship codes. In this case they are used as modules with only a partial data encapsulation. Further work is ongoing on these libraries to improve the encapsulation and the APIs in order that they can be used also outside of the original codes.

The compute intensive functionalities are now performed by domain specific mathematical libraries as for example FFTXlib, SpFFT for 3D FFT operations or LAXlib, DBCSR and COSMA for parallel linear algebra. This has allowed to improve out performance portability and has streamlined the porting to other HPC architectures concentrating the effort on the specific libraries.

We are working on this side to extend the compatibility between the APIs of the different mathematical libraries and the portability to other architectures and backends.

The MAX library set now provides also a full fledged utility layer meant to provide the programmers with a general architecture agnostic backbone for managing parallelism, I/O, error handling, and timing. During these 18 months the necessity has also emerged to include in the utilities an abstraction layer for masking all those operations needed in heterogeneous architectures for the movement and the synchronization of the data between the host device memory spaces. For this reason a new library, `DevXlib` has been planned and is currently under development. A significant part of the effort in the development of this library in the next months will be put in enabling the usage of different backends.

Acronyms

ESL The Electronic Structure Library [8]. 4

HPC High Performance Computing. 19

SDP Software Development Plan [1]. 4–6, 19

XSD XML Schema Defininition [2]. 9

References

[1] Baroni, S. *et al.* First report on software architecture and implementation plan. Deliverable D1.1 of the H2020 CoE MaX (final version as of 30/03/2019). EC grant agreement no: 824143, SISSA, Trieste, Italy. (2019).

[2] https://www.w3.org/standards/xml/schema.

[3] Electronic structure library coding workshop: Drivers. trieste, july 10-21 2017. https://gitlab.e-cam2020.eu/esl/ESLW_Drivers.

[4] Edwards, H. C., Trott, C. R. & Sunderland, D. Kokkos: Enabling manycore performance portability through polymorphic memory access patterns. *Journal of Parallel and Distributed Computing* **74**, 3202 – 3216 (2014).

[5] Zenker, E. *et al.* Alpaka - An Abstraction Library for Parallel Kernel Acceleration (IEEE Computer Society, 2016). URL http://arxiv.org/abs/1602.08477. 1602.08477.

[6] Matthes, A. *et al.* Tuning and optimization for a variety of many-core architectures without changing a single line of implementation code using the alpaka library (2017). URL https://arxiv.org/abs/1706.10086. 1706.10086.

[7] Hornung, R. *et al.* Asc tri-lab co-design level 2 milestone report 2015 (2015).

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