Ref. Ares(2023)8881593 - 27/12/2023

D3.1

Interim report on performance analysis of MAX software

F. Affinito, L. Bellentani, J. Javoršek, B. Krašovec, P. Delugas, A. Garcia, L. Genovese, I. Girotto, M. Ippolito, J. Gutiérrez Moreno, L. Riha, N. Spallanzani, D. Wortmann

Due date of deliverable: 31/12/2023 (month 12)
Actual submission date: 27/12/2023
Final version: 27/12/2023

Lead beneficiary: CINECA (participant number 7)
Dissemination level: PU - Public

Document information

Project acronym:	MAX
Project full title:	Materials Design at the Exascale
Research Action Project type:	Centres of Excellence for HPC applications
EC Grant agreement no.:	101093374
Project starting / end date:	01/01/2023 (month 1) / 31/12/2026 (month 48)
Website:	www.max-centre.eu
Deliverable No.:	D 3.1

Authors: F. Affinito, L. Bellentani, J. Javoršek, B. Krašovec, P. Delugas, A. Garcia, L. Genovese, I. Girotto, M. Ippolito, J. Gutiérrez Moreno, Lubomir Riha, N. Spallanzani, D. Wortmann

To be cited as: F. Affinito et al., (2023): Interim report on performance analysis of MAX software. Deliverable D3.1 of the HORIZON-EUROHPC-JU-2021-COE-01 project MAX (final version as of 27/12/2023). EC grant agreement no: 101093374, CINECA, Consorzio Interuniversitario Cineca.

Disclaimer:

This document's contents are not intended to replace consultation of any applicable legal sources or the necessary advice of a legal expert, where appropriate. All information in this document is provided "as is" and no guarantee or warranty is given that the information is fit for any particular purpose. The user, therefore, uses the information at its sole risk and liability. For the avoidance of all doubts, the European Commission has no liability in respect of this document, which is merely representing the authors' view.

Versioning and contribution history:

Version	Date	Author	Note
1	01/12/2023	F. Affinito, L. Bellentani, J. Javoršek, B. Krašovec, P. Delugas, A. Garcia, L. Genovese, I. Girotto, M. Ippolito, J. Gutiérrez Moreno, Lubomir Riha, N. Spallanzani, D. Wortmann	First draft
2	02-14/12/2023	M. Ippolito, N. Spallanzani, L. Bellentani, D. Wortmann	Updates and integrations
3	14-24/12/2023	A. Ferretti	Integrations and final polishing

D3.1 Interim report on performance analysis of MAX software

1 Executive Summary

WP3 takes care of the continuous assessment and analysis of the parallel performance of the MAX flagship codes, pointing up the direction for the development aimed at the effective exploitation of the existing technology. To this aim, we make use of tools for code profiling and optimisation and of the most recent programming models. WP3 activity is functional to many other tasks in different work packages (WP1, WP2, WP4, WP5), in order to provide feedback on the progress obtained in terms of performance enhancement with respect to the relevant metrics and to the parallel efficiency. On the other hand, an important output of this WP is to discover and monitor code bottlenecks, to identify the code or architecture feature responsible for them (memory bandwidth, communication imbalance, latency, bandwidth to GPU, etc), and to propose dedicated solutions also through the engineering of ad-hoc proof-of-concepts. The solutions of these bottlenecks that require code refactoring or replacement of an algorithm will be implemented within WP1 by the code developers. Our activity will therefore be in continuous synergy with the developing teams of the MaX flagship codes.

In the following we will describe how we have organised the benchmarking and profiling activity, introducing the tools adopted (Section 3) and presenting some preliminary results (Section 4, 5 and 6). The deployment status of flagship applications on EuroHPC systems will also be reported (Section 7).

2 Introduction

In order to improve uniformity and homogeneity in the data produced for each MAX code and to have common metrics to evaluate the code's performance, we adopted a common strategy for benchmarking and profiling activities of all MAX codes. In this regard, we have decided to use the same benchmarking and profiling tools for all MAX codes, and, to coordinate these activities, we organise periodic meetings between CINECA experts and each MAX code community. These side-by-side meetings should have the purpose of continuously evaluating the code performances, optimising them, identifying possible bottlenecks and finding solutions, but also to train MAX code developers on benchmarking and profiling tools by gaining experience on these. Another outcome of this close collaboration between CINECA experts and MAX code developers will be that CINECA will acquire a more in-depth knowledge of all MAX codes, and this will allow CINECA experts to be able to provide ever better support for these codes.
Leonardo[1], the pre-exascale Tier-0 EuroHPC supercomputer hosted by CINECA, will be the preferred system for the profiling and benchmarking activities being the largest HPC architecture available within MAX and being accessible to all MAX code developers. However, within the CASTIEL[2] project - promoting interaction and exchange between National Competence Centres - we should soon have access to other

[1] https://www.hpc.cineca.it/systems/hardware/leonardo/
[2] https://eurohpc-ju.europa.eu/research-innovation/our-projects/castiel_en

EuroHPC architectures. Then, MAX codes will be ported and deployed on all the architectures we will have access to, addressing possible issues and working towards their resolution.
To run benchmarks within MAX, we adopted JUBE[3], which is a tool developed at Julich Supercomputing Center that provides a script-based framework to easily create benchmark sets, run them on different computer systems and evaluate the results. For each benchmark application, the benchmark data is written in a certain format that allows JUBE to obtain the desired information. The data analysis can be automated by pre- and post-processing scripts extracting information and storing it, so it can be easily retrieved for visualisation.
To facilitate the adoption of JUBE as the official tool for benchmarks within MAX , in collaboration with WP5, CINECA organised a webinar dedicated to jube followed by a mini-hackathon. The webinar documentation with related examples are available in the JUBE4MAX[4] repository, that is the repository for MAX JUBE material.
The first phase of the project was dedicated to the porting of the MAX codes to Leonardo and to the preparation of JUBE scripts; subsequently, we worked on their benchmarking and profiling on the cluster. Some of the preliminary results obtained so far are reported in Sections 4, 5 and 6.

3 Methodology

3.1 Benchmarking and Profiling

Monitoring performance analysis and having a continuous evaluation of scalability to provide information to the code developers are the main objectives of WP3. Achieving this goal required in first place the identification of a flexible approach to benchmarking and profiling, which ensures a straightforward exploitation of the results through the standardisation of the output formats.
The requirement of flexibility is determined firstly from the object of our analysis: a selection of materials science programs with different command-line parameters, such as executable arguments, inputs, dependencies, etc. A comprehensive approach to analyse such different codes demands for an application-agnostic *interface,* that is able to switch from measuring one code to another with minimal modifications from the user point of view.
Considering the variety of the available HPC hardware, our method should also streamline the comparison of performance measurement on different clusters. This is of particular relevance to support the effort of other WPs on code portability, which would be fostered by a platform-agnostic ecosystem, able to characterise the application on different machines with comparable instructions and tools.

[3]https://www.fz-juelich.de/en/ias/jsc/services/user-support/software-tools/jube?expand=translations,fzjsettings,nearest-institut
[4] https://gitlab.com/max-centre/JUBE4MaX/-/tree/develop?ref_type=heads

Moreover, in the scientific domain covered by MAX codes, the size of the workload can not be increased straightforwardly, but depends on the complex interplay between physical and mathematical input parameters. To analyse the response of an application to increasing workloads with physical significance, our ecosystem should then provide an organised *library* of production-like test cases with their input files, to be loaded at "runtime" by a workload-agnostic interface.

In our view, benchmarking and profiling share the same methodological background. Both activities require (i) launching the simulation with different parameters (e.g. the number of nodes), (ii) parsing data from output files, either the logfile of the application or the summary in the profiling report, and (iii) analysing the trend of the parsed metrics with tables.

The main operational difference in profiling is the addition of a command line interface before the application executable, or the use of an instrumented executable to activate the measuring *daemon*. Command switches, environment variables, or additional input files might be needed at runtime to configure the instrumentation, for example to specify the events to collect (MPI, CUDA, OpenACC runtime, etc) or filter routines out. Their definition differs according to the tool adopted, e.g. Score-P, Nsight Systems, TAU, Rocprof, etc. Moreover, the preference of a profiling tool depends on various factors, including the availability and suitability of the tool on a platform, the kind of events to measure, the level of support for large scale analysis, the software stack for the application build, instrumentation conflicts, etc.

Thus, the overhead of the measurement in a performance engineering workflow requires the continuous interaction between parties with different expertise. The developer's contribution is essential to identify the most relevant test cases to focus on and to clarify the application algorithm, while the one of the HPC specialists is to set up the measurements and interpret the trace and profile results. Figure 1 shows a schematic representation of the performance engineering workflow and the role of the developers and HPC specialists in the different phases.

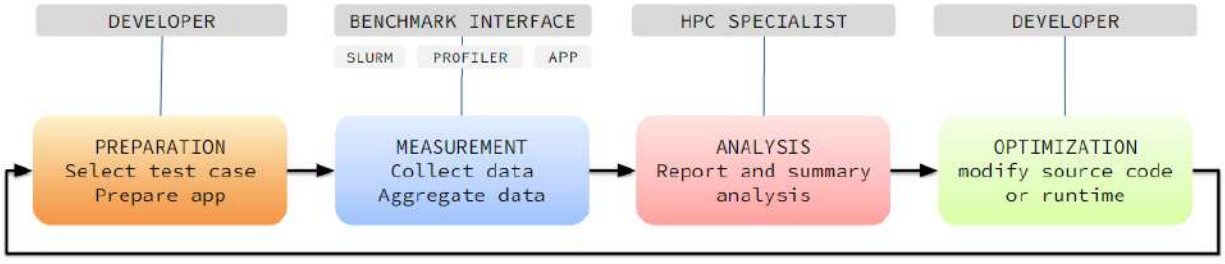

Fig. 1. Performance analysis workflow; the use of a benchmarking interface streamlines the measurement phase by hiding the lower level details of the profiling to the developer.

In our approach, we wish that all the degrees of freedom related to the choice of a particular tool are additional parameters selected from a dedicated library at runtime, according to generic user

instructions. The same mechanism, that we apply to platform-dependent or application-dependent parameters while benchmarking, should thus be extended to profiling-dependent ones. This way, the specific syntax of command switches and specifications for a given profiling tool are hidden to the user, which can generate summaries and traces more independently, thus fostering the performance engineering workflow.

3.2 Standardisation of the outputs

Together with the measurement phase, our approach should streamline the storage of results and the standardisation of the outputs. MAX applications enclose inner clocks to measure the time to solution of the whole simulation or a particular phase of it, while profiling summaries supply additional metrics to better characterise application performance. However, this information is usually displayed in a non uniform way, depending on the software or the application, and their trend with respect to the simulation parameters is not directly available.
Our interface should then be able to collect selected metrics by parsing different kinds of log files, produced by the application or the profiler, analyse them statistically in case of multiple occurrences, and print out their evolution with respect to the benchmarking parameter. The homogeneity in the presentation of results is crucial to preserve the history of code optimization, compare the performance between different codes and add on top of it tools for visualisation, e.g. gitlab pages.

3.3 JUBE for the agnostic benchmarking ecosystem

So far, we discussed the requirements for the interface in the frontend, which should be as general as possible for the user launching the set of simulations, i.e. it should not depend on the application, platform, workload, and profiling tool chosen.
At the bottom of this interface, our ecosystem should provide a *library* of scripts tailored for the (i) MAX codes, (ii) EuroHPC platforms or local clusters, (iii) production-like use cases, and (iv) profiling tools. This library translates the general directives given by the user to the frontend interface into software, platform, workload or profiling specific parameters needed to actually run the set of simulations. For example, the library of platforms should take care of setting the options for a job script on a given machine, according to the scheduler available, the number of cores, sockets and GPUs per node, with respect to the resources requested for the benchmark. These circumstantial details are defined in specific scripts, curated by experts in the domain of the *library* itself, in accordance with the principle of *separation of concerns* driving MAX actions.
While the implementation and maintenance of the library should be handled separately by the experts in the topics covered by the library itself, our approach envisages regular face-to-face meetings between developers and HPC specialists to exchange information for and from the performance analysis. Coherently with the performance engineering workflow depicted above (Fig.1), these meetings aim in

particular to (i) identify a suitable test case and the main metrics to investigate for the performance assessment, (ii) provide details about the algorithms and history of the code that are helpful to understand the profiles and the reports, (iii) provide feedback about possible directions of optimization to implement.

Coherently with the requirement detailed above, we identified JUBE as the preferred tool to support benchmarking and profiling activities of WP3. JUBE automates benchmarks and standardises the outputs, which is important for reproducibility and hence comparability of the results. Under this respect, JUBE can help in performing and analysing benchmarks in a systematic way, and it allows custom workflows to be able to adapt to new architectures. This tool provides a script based framework, to easily create benchmark sets, run those sets on different computer systems and evaluate the results. One key advantage of using JUBE, as opposed to manually running an application in different configurations in a job script, is that individual run configurations are automatically separated into different *workpackages* with individual run directories, while common files and directories (like input files, preprocessing, etc.) can easily be integrated into the workflow. Moreover, JUBE's commands can cover all actions common to a benchmarking session. For example, `jube run` executes a number of commands populating the *workpackages* and submitting the job scripts. `jube analyze` parses the log files and collects the result of the benchmark, while `jube result` organises the data parsed across *workpackages* in tables.

Two different types of input formats are supported by JUBE: XML-based files and YAML-based files. The former has been chosen for MAX's benchmarking environment scripts; however, thanks to the interoperability between the two languages, there is no severe restriction on this choice.

3.3.1 The JUBE4MAX ecosystem of scripts

Figure 2 shows a schematic representation of the steps needed to implement a benchmark with JUBE. The first 2 steps are related to the preparation of the inputs to the JUBE commands. Following the approach described in Sec. 3.1, **application.xml** is the input containing all the parameters related to a given application, e.g. the name of the executable and its arguments. This script is maintained by the developers, and there will be one for each application addressed by the MAX project.

Differently, **platform.xml** is curated by HPC centres and contains the parameters related to the machine, to be derived from the resources allocated by the benchmark sets, from the scheduler ones such as the SLURM parameters, to the binding options. We envisage to prepare one platform.xml for each EuroHPC architecture to which we will have access; currently, inputs for Leonardo, LUMI and MareNostrum-4 (soon upgraded to MareNostrum-5) are available, among other clusters.

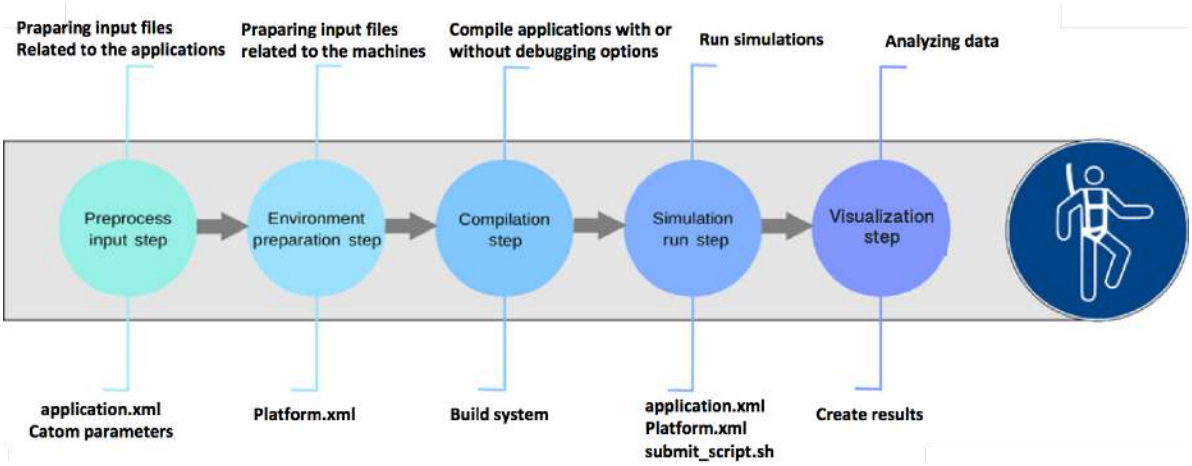

Fig. 2. Schematic overview of the steps in a JUBE based run.

In addition to these fundamental scripts, the benchmark requires a **workload.xml** file providing parameters and location of the inputs for the application itself. Finally, if the session involves profiling, a **script.xml** file loads the wrappers needed for instrumentation. Application.xml and script.xml files will contain also the patterns of the metrics to collect from the application log file and the summary, respectively, as well as the design of the final result tables.

Once the JUBE inputs are ready, we can move on to the next steps; we can compile the applications with optimised options for benchmarking and production runs and with debug options for debugging and profiling activities. Then, we can run the applications and get the results to analyse.

MAX JUBE scripts detailed above are archived in the JUBE4MAX repository[5]; its structure is depicted in Fig.3. INTERFACE contains the scripts with parameters specific to the benchmark or the profiling session, such as the number of tasks on which the scalability is analysed. These scripts should be as general as possible and not to depend on the application, profile tool, or workload addressed by the benchmark (an example can be found at the following URL[6]). The remaining folders organise the scripts and the input according to the concern. In particular, PLATFORMS and APPLICATIONS folders contain respectively the platform.xml and applications.xml scripts, while WORKLOAD folder contains the inputs for the MAX applications. Finally, the SCRIPT folder contains wrappers needed for the measurement or post processing, to be filled according with the information in a related xml file.

Note that JUBE4MAX is a public gitlab repository, as well as the benchmark repository with the results, and we are open to external contributions to increase the support of more platforms and tools. In addition to supporting the activities of other MAX WPs and the optimisation of our codes, we hope that

[5] https://gitlab.com/max-centre/JUBE4MaX/-/tree/develop/max-inputs?ref_type=heads
[6] https://gitlab.com/max-centre/JUBE4MaX/-/blob/develop/max-inputs/interface/common/benchmark.xml?ref_type=heads

our methodology and the resulting ecosystem of scripts can be of service also to other CoEs, HPC centres and users. Thanks to the interoperability between platform, application, script xml files, the components of our JUBE4MAX ecosystem could be exploited also partially to benchmark more applications on the supported platforms or MAX applications on new platforms, while general users could enrich our library of test cases with more production workloads.
In order to facilitate the collaboration among different parties to the scripts in our repository, we will maintain a version controlled documentation based on open source utilities such as sphynx. There, we will discuss the context of the different xml files implemented for our ecosystem, the main commands of JUBE and how to contribute to the project.

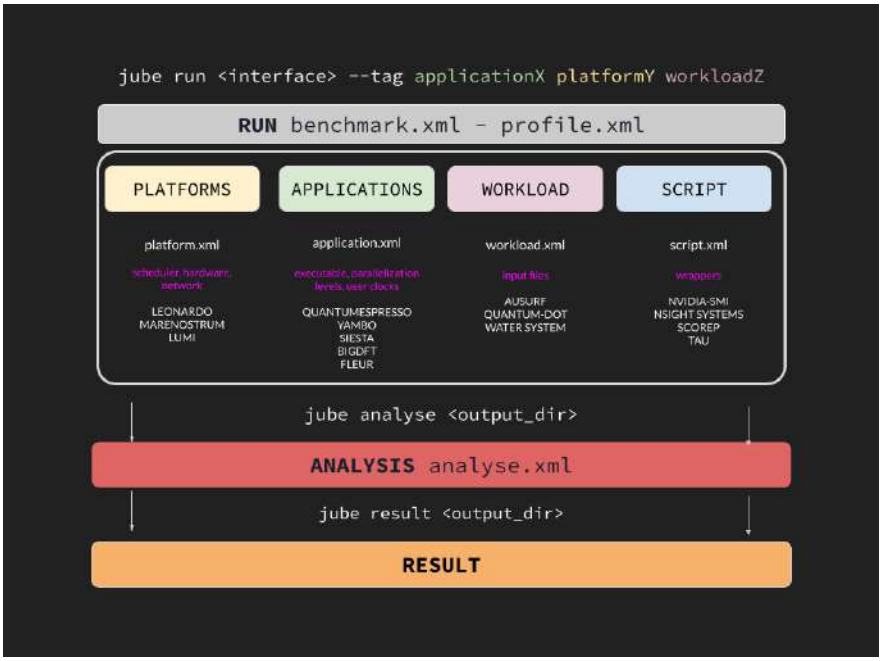

Fig. 3. Steps in a JUBE based benchmarking/profiling session and JUBE4MAX repository structure.

3.3.2 The support to workflow

Our ecosystem of JUBE scripts will also support the benchmarking of single steps in code workflows. This is important for MAX codes such as YAMBO and PHONON from QuantumESPRESSO, whose execution requires simulating the ground system wavefunctions with PWscf (that is the core component of the QuantumESPRESSO suite). In principle, only the output directory from PWscf is needed; however, this information can be quite consuming to be stored online and it is usually not backed up. To preserve the reproducibility of the simulation without adding consuming information, our JUBE scripts are extended

to workflow as in this example[7], where the benchmark is limited to a given part of the code chain, while the full provenance of the simulation is stored.

3.3.3 Results storage & visualisation

Fig. 4 shows how JUBE organises the outputs of a benchmark. Each session launched with the command `jube run` generates a benchmark directory with a unique id. Within the benchmark folder, JUBE creates a separated directory (a *workpackage*) where a set of operations (a series of *steps*) are executed for one of the combinations of parameters addressed in the benchmark. In each *workpackage*, we preserve inputs, output and job files so that the simulation can be independently reproduced, given the availability of the executable.

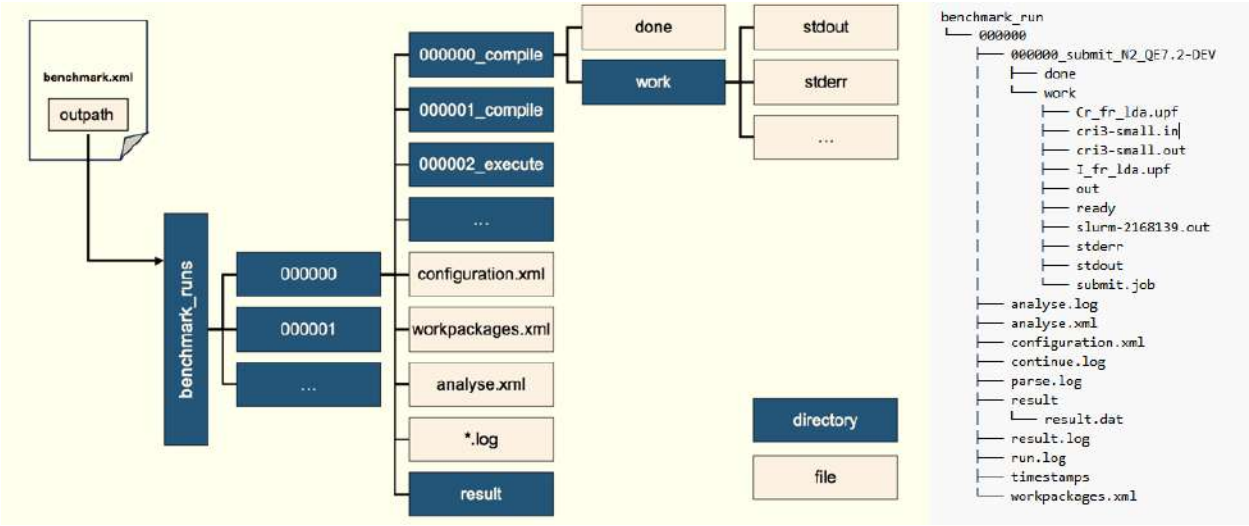

Fig. 4. Directory tree of the Jube output.

In the root directory of the different *workpackages*, JUBE stores logfiles recording the commands executed, and a number of .xml files with the different concretizations of the parameters sets used in the benchmark or analysis. Finally, the results of the analysis phase across the different parameter combinations are stored in a file with uniform format (csv). One example of output is reported in the following snapshot.

```
Nodes;Tasks;install;electrons;cbands;diaghg;Xegterg;vloc_psi;h_psi;sum_band;walltime;vloc_calls
1;4;QE7.2-DEV-AWARE;59.37;46.75;10.60;39.34;13.61;19.42;8.45;70.01;284
2;8;QE7.2-DEV-AWARE;55.97;45.04;10.73;41.04;17.99;23.50;8.06;63.75;276
```

[7] https://gitlab.com/max-centre/JUBE4MaX/-/blob/yambo-jube/max-inputs/workflow-yambo.xml

```
4;16;QE7.2-DEV-AWARE;56.67;46.03;11.69;43.38;20.59;25.76;8.33;64.69;284
6;24;QE7.2-DEV-AWARE;57.45;47.72;12.44;45.58;18.09;24.63;7.80;67.24;297
8;32;QE7.2-DEV-AWARE;86.53;71.38;11.01;69.46;42.20;48.92;13.30;96.74;282
```

In perspective, the storage will ensure reproducibility while the data homogeneity will permit to standardise visualisation of results across the different applications. This simplifies the implementation of scripts, e.g. based on python, or jupyter notebooks for data plotting. The benchmark results will be stored on an open dedicated repository; the common format of the outputs streamlines the possibility to integrate this repository with gitlab pages for data disseminations to users.

4 MAX applications status on Leonardo

As mentioned in the Introduction, the pre-exascale Tier-0 EuroHPC supercomputer Leonardo hosted by CINECA will be the preferred system for profiling and benchmarking activities. Leonardo is indeed the largest HPC architecture available within MAX and accessible to all MAX code developers. Leonardo consists of two main partitions: Booster Module and Data-centric Module. The Booster module partition was the only one available so far, and it is based on the BullSequana XH2135 supercomputer nodes, each with four NVIDIA A100 Tensor Core GPUs with 64GB of memory and a single Intel IceLake CPU.
However, going forward with the project within CASTIEL, we should also have access to all other EuroHPC architectures, where all MAX codes will be ported and deployed. We expect to have at least one benchmark campaign per year on all the other architectures to which we will have access. Having adopted JUBE as the official tool for MAX benchmarks should make it relatively straightforward to replicate the same benchmarks on the other platforms.
Since the production phase of Leonardo only started in September, in the initial phase of MAX we used M100, based on IBM Power9 architecture and Volta NVIDIA GPUs. M100 was used to identify the most suitable software stack for each code and to test the various tools for benchmarking and profiling activities.
When Leonardo was ready for production, the first step was the porting of all the MAX codes, finding the most suitable software stack for each one. Furthermore, the application.xml files for each MAX code and the platform.xml file for Leonardo were prepared and tested.
Each MAX code was compiled on Leonardo in an optimised way for benchmarking purposes and for production runs and the executables with debug options for profiling purposes were also obtained. The optimised executables will also be available to external users, while the executables for debugging and profiling purposes are for use only by participants in the MAX project and are in a storage area dedicated to the project.

The status of the MAX flagship codes on Leonardo is summarised in the following table.

APPLICATION NAME	Installed	Compilers	Jube Scripts	Benchmarks	Profiling Meetings
SIESTA 5.0.0	Yes •tested •executable for the pofiling	Gcc11.3.0-Openmpi4.1.4	Application Tested on Leonardo	Covid-12k	2
YAMBO 5.2.0	Yes •module •executable for the pofiling	Nvhpc23.1-Openmpi4.1.4	Application Tested on Leonardo	GrCo	4
Quantum ESPRESSO 7.2	Yes •module •executable for the pofiling	Nvhpc23.1-Openmpi4.1.4	Application Tested on Leonardo	Ausurf (Pw) Cnt10por8 (Pw) Si (Ph)	2
BIGDFT	Yes •tested •executable for the pofiling	Gcc11.3.0-Openmpi4.1.4	wip	wip	2
Fleur	Yes •tested	Nvhpc23.1-OpenMPI4.1.4	Application Tested on Leonardo	wip	

Table 1. Status of the MaX flagship codes on Leonardo.

The latest versions of Siesta, Yambo, Quantum ESPRESSO, and BigDFT have been installed and tested on Leonardo. Yambo, QuantumESPRESSO, and Fleur were installed using the Openmpi/4.1.4 and Nvhpc/23.1 compilers, while Siesta and BigDFT were installed with the Openmpi/4.1.4 and gcc/11.3.0 compilers.
Yambo and QuantumESPRESSO were installed as modules with Spack, the modules for the other codes will soon be available. The executables obtained with debugging options, to be used for profiling activities, are also available on Leonardo in the MAX storage area, accessible only to MAX staff .

4.1 QuantumESPRESSO

The accelerated version of QuantumESPRESSO (QE) has been integrated in the module environment of Leonardo since the beginning of the production phase. Regarding the JUBE ecosystem, QE-specific application xml files are available for pw.x (PWscf code) and ph.x (PHONON code), but can be easily extended to the other codes of the suite. Most of the benchmarks and profiling presented here have been obtained by exploiting JUBE scripts at different levels of maturity.

Regarding the performance, early benchmarks have been done on PWscf and PHONON. The first is the most historical code of the suite, and its boost on GPUs has been largely addressed during previous MAX editions together with the Car Parrinello code. The second code has been accelerated only recently with the preferred directive-based approach, where OpenACC is added in the higher level of the code to improve maintainability, while CUDA-Fortran remains in the lower level libraries to ensure performance. A recent article[8] demonstrates the speed up provided by accelerators with respect to the use of CPU only, and analyses the performance of the codes on heterogeneous platforms based on NVIDIA-GPUs for the main distribution schemes (Selene and the dismissed Marconi100).
Beyond pool distribution, whose efficiency for accelerated runs has been largely addressed, the interaction of other parallelization levels with accelerated hardwares has been less investigated.

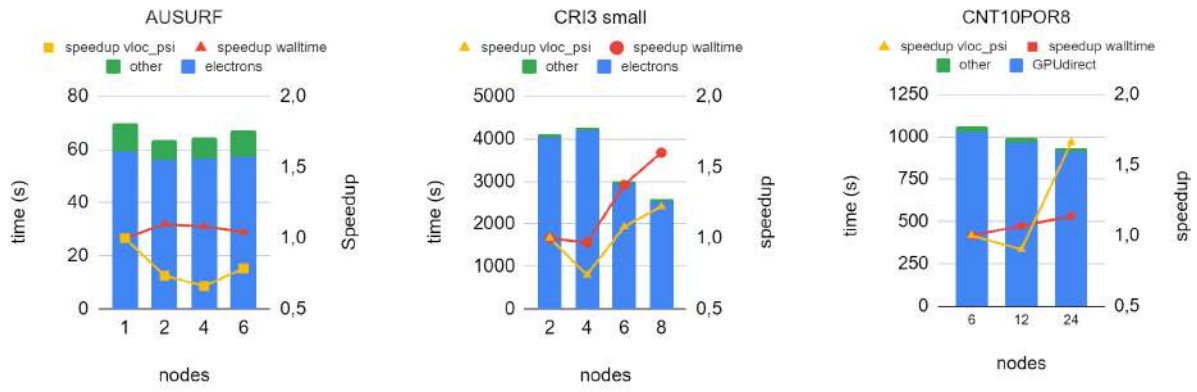

Fig. 5. Comparison of the time to solution (stacked columns) of electrons (blue section) plus other routines (green section) for three samples. The dotted lines are the efficiency of the (red) total runtime and (yellow) vloc_psi, the main routine of the driver for distributed Fourier Transform, respectively). Details on the workloads are reported in Tab.2. The QE version is the develop from 7.2 with GPUdirect on Leonardo.

In particular R&G distribution, which is the default parallelization layer in the CPU version of QE, has demonstrated in the article mentioned above limited scalability. R&G distribution, which improves latency times on parallel homogeneous nodes, impairs instead on GPUs the overall throughput because of the high amount of communications triggered. A deeper characterisation of the issue beyond time to solution is still missing. Further, the advantage in using GPUdirect RDMA communications with respect to the alternative algorithm staging data through the CPU has not been assessed.
This analysis is of particular relevance for the simulation of memory demanding workloads which can not fit the memory of one node, thus requiring R&G distribution with inter-node communications. Figure 5 confirms the absence of scalability for this distribution also on Leonardo, by comparing the time to

[8] I. Carnimeo et al., Journal of Chemical Theory and Computation 2023 19 (20), 6992-7006

solution and the efficiency of the total simulation and the main kernel for distributed FFTs (vloc_psi) of three workloads with a different memory footprint: AUSURF, CRI3 small and CNT10PORr8[9].

NAME	FFT dimension	G-vectors	Kohn-Sham states	k-points
AUSURF	125, 64, 200	100747	800	2
CRI3 small	120, 192, 640	904069	1944	3 X 2 spins
CNT10POR8	375,375,375	2443081	3139	1

Table 2. Physical parameters of the three workloads benchmarked in Fig.5.

The band group parallelization, distributing the wavefunctions instead of their components, seems more suitable for the throughput approach needed for heterogeneous parallelism. This happens because each wavefunction is assigned as a whole to a single GPU, and the 3D FFTs are performed on it without communication overhead. The band parallelism poses though other optimisation problems, such as in terms of load balancing, memory footprint and communication. We have started working at this point by analysing the current band-group implementation. This analysis is reported in Section 5, and already provides a useful input for planning the refactoring of band group parallelism which will be taken in charge jointly by WP3 (tasks T3.1 and T3.2) and WP1. First point of this work will be the preparation of two mini-apps emulating the iterative solver and the blocked Gram-Schmidt orthogonaliser in pw.x.

In addition to band and pool distribution, QE exposes images, an additional MPI group to distribute independent calculations. This distribution layer is available only in some codes, such as NEB and PHONON, and is an ideal candidate to achieve large-scale occupancy on heterogeneous hardware, because of the little amount of communications entailed. Usually, it is more prone to performance degradation due to workload imbalances or serial parts.

To assess the potential of image distribution on accelerators, the system presented in the above-mentioned article (Ref. [7]) has been recently benchmarked on Leonardo up to the largest scale. The application can distribute k-points to 64 pools maximum, and the calculation of the contributions to the dynamical matrices by the different irreducible representations to 192 images maximum. This calculation might occupy 3072 nodes, corresponding to 3072*4=12288GPUs, almost the full Leonardo Booster partition. At the time of the benchmark, we did not have the possibility to reserve such a large number of resources and so we were able to demonstrate the scalability only up to 1024 nodes, corresponding to 1024*4=4096 GPUs.

Due to the little amount of MPI communications, the scalability of the main kernel, solve_linter, is well sustained up to 1024 nodes. At this scale, the phonon simulation might be affected by imbalances

[9] Inputs are available here https://gitlab.com/max-centre/JUBE4MaX/-/tree/main/max-inputs/workloads/qe/pw/

between the load of each task and serial part, the latter being further exposed by the acceleration of GPUs. We stress that PHONON simulates a workflow composed by steps that scale on different distribution schemes, or are characterised by different workloads.

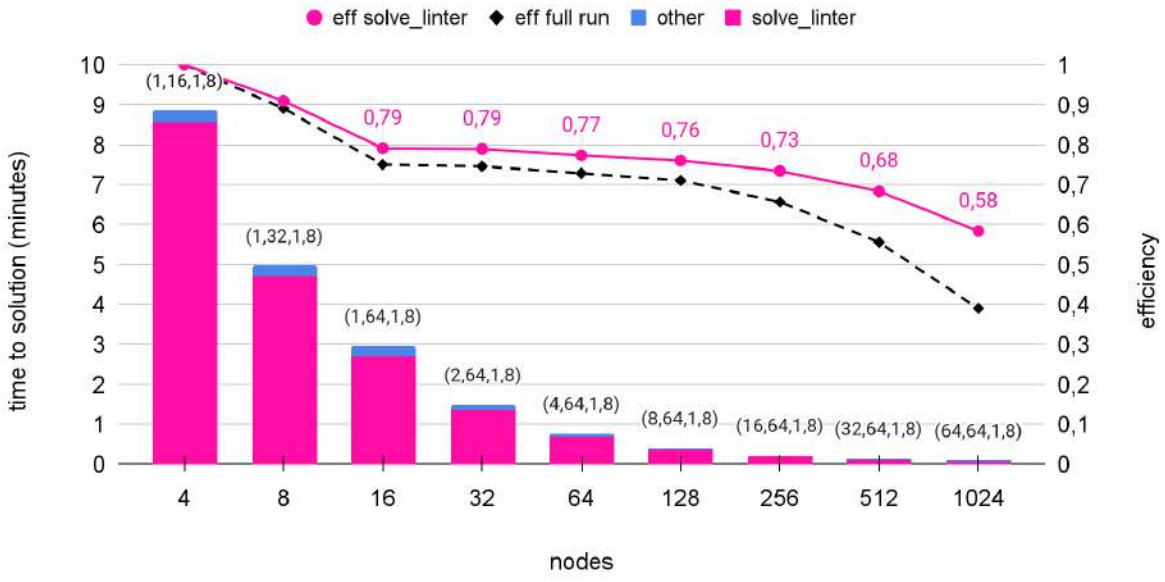

Fig. 6. Walltime of the PHONON runs on Leonardo (columns). Most of the time is spent in the `solve_linter` kernel (purple section) to compute the contributions to the dynamical matrix by an irreducible representation. The system is scaled on pools up to 64 GPUs, which is the largest number of pool processes available for this workload; further scaling is achieved up to 64 images. We could in principle reach 192 images, but we could not further scale the application due to the limited availability on Leonardo's booster partition at the time of the benchmark. The numbers between brackets on top of the columns represent the parallelization adopted, according to the notation *(ni,nk,npw,omp)*, where ni is the number of images, nk the number of pools, npw the number of R&G processes and omp the number of openmp threads. The black dashed line is the efficiency of the whole simulation, while the purple continuous line refers to the efficiency of solve_linter routine.

At the largest scale, this might induce a significant waste of resources due to time spent in synchronisation barriers or unnecessary replicas in the calculation. In particular, at the beginning of the linear response simulation, PHONON implements a non self consistent step to compute the wavefunctions in the perturbed system, starting from the ground-state output of PWscf. In a phonon frequency calculation, this step is implemented by all images independently, despite the fact that they are calculating the same physical quantity. The replica avoids potentially dangerous master-slave patterns, but might have a significant cost at large scale if the time spent in the replica is comparable with the time spent per image in the distributed calculating. To face this, we should consider, in

collaboration with WP2, the integration of the phonon code into a workflow manager, which executes these two phases separately with tailored resources, thus minimising the time spent in synchronisation barriers.

4.2 Yambo

Following the structure discussed in Section 3.5, we created an XML file in order to characterise the Yambo application and a workload labelled *GrCo*[10]. The workload is a complete GW Yambo workflow for a Graphene/Cobalt interface composed by a graphene sheet adsorbed on a cobalt slab (four layers thick), and a vacuum layer large as the cobalt slab.
Then, a benchmark XML script has been developed to customise certain Yambo runtime features. For instance, the script allows for the specification of MPI parallel levels utilised by Yambo in various drivers. This is achieved through specific script variables initialised by an expression function of the total tasks, and resolved by the JUBE Python backend. Consequently, when conducting a scalability test, the script populates the Yambo input files of every run with the appropriate variables and values for the desired parallelization strategy.
Usually, a benchmark used for a scalability test is not suitable for use as a profiling test. This is because the overhead introduced by the profiler tracing functions may become excessive. This issue can be resolved by manipulating specific input variables that allow for the reduction or increase of computational workload according to the requirements. The same variables were added in the benchmark XML script that now can be used also for managing a profiling test.
Another issue that we faced is that the ground-state starting databases needed by Yambo have to be already available in the cluster and the path has to be provided to the script. Infact, in order to work, Yambo needs to rely on the Kohn-Sham wavefunctions previously calculated by a DFT procedure, in our case using QuantumESPRESSO. To this aim, we created a workflow script that exploits the step feature provided by JUBE. Some new variables are introduced in the script to tailor also the size of the ground-state calculation. The complete workflow consist on five steps: (i) SCF calculation and (ii) NSCF calculation both done using PWscf; (iii) P2Y calculation to convert QuantumESPRESSO output in Yambo databases; (iv) initialization of the databases with the Yambo executable; and finally (v) the benchmark or the profile of the Yambo workload. Using the right combination of JUBE tags it is possible to choose whether to run the complete workflow or only the final step providing the path of the input databases.
As mentioned, the JUBE tool is useful not only for benchmarking, but also for profiling the code. This can be done thanks to the possibility of wrapping the calculation at runtime via some script available in the JUBE4MAX repository. For Yambo this is a work in progress.

[10] https://gitlab.com/max-centre/JUBE4MaX/-/tree/main/max-inputs/workloads/yambo/grco?ref_type=heads

In Figure 7 (top panel) it is reported the scalability test performed on the GrCo workload tailored having in mind that a profiling test will be done on the first run with an efficiency lower than 60% (in this case the run at 160 MPI tasks, 40 nodes).

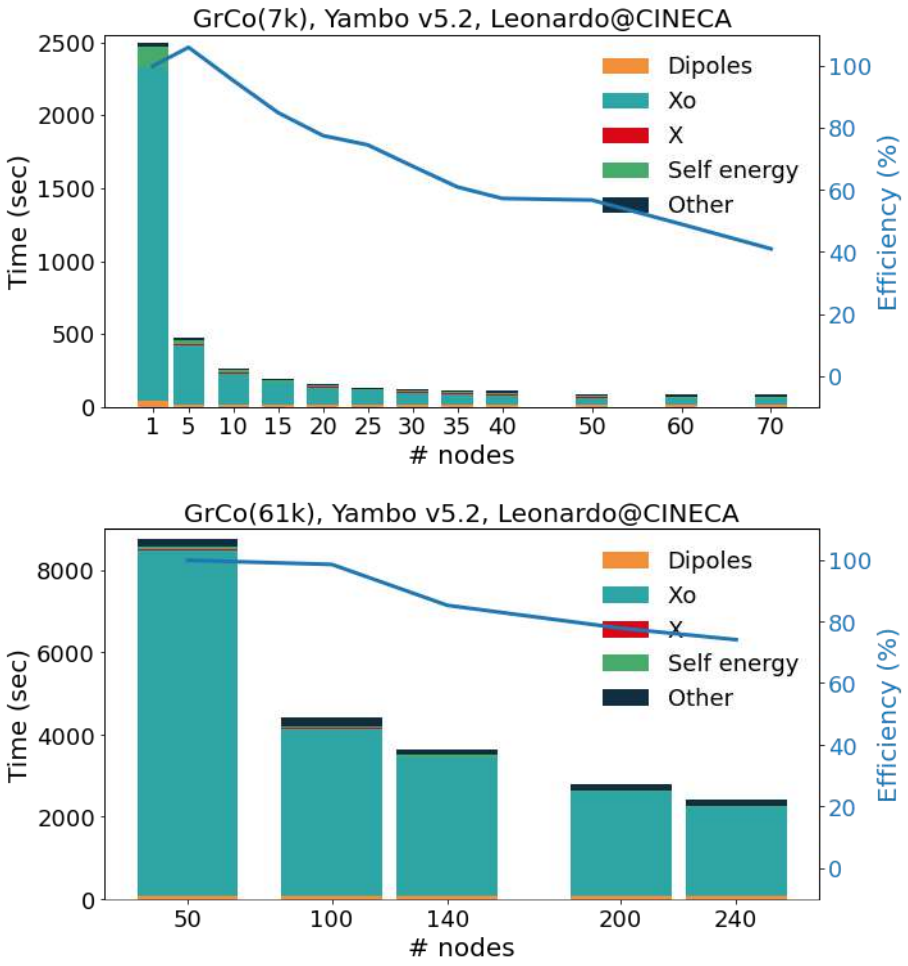

Fig. 7. Yambo strong scalability tests performed using JUBE and the GrCo workload on Leonardo-Booster partition (CINECA). Top panel: small system with 7 k-points, 2000 bands, 20 Ry Xo cutoff. Bottom panel: medium-size system with 61 k-points, 2000 bands, 20 Ry Xo cutoff.

This is why the system is relatively small compared to usual Yambo scalability tests. In the ground-state calculation we set a k-point grid 6x6x1 that leads to only 7 total k-points. In the GW workflow the total number of bands used was 2000 and the energy cutoff on the irreducible response function (Xo) set to 20 Ry (both production values) in order to guarantee a good scalability to a discrete amount of nodes.

In addition to that, we also performed a strong scalability test of a larger GrCo benchmark (Fig. 7 bottom panel), starting from a ground-state calculation with a 24x24x1 k-point grit, that means a total of 61 irreducible k-points. The GW workflow uses the same parameters of the previous one (2000 bands and 20 Ry Xo cutoff). In this case we can see a very good scalability up to 240 nodes, corresponding to 240*4=960 GPUs. Eventually, we will schedule larger runs with an agreement of CINECA user support.

4.3 Siesta

JUBE scripts for Siesta are relatively mature and allow for easy control of the system size and quality parameters of the workloads (e.g. basis-set size, mesh cutoff, supercell size), as well as the operational parameters such as number of nodes, tasks per node, and optional profiling options.
The strategy followed in Siesta for GPU acceleration rests on the fact that the solver stage typically accounts for the lion's share of the total computational time, and in the availability of accelerated solver libraries that are portable to various architectures. Hence performance improvements and extensions of functionality are immediately available to the code on different platforms. We use the ELPA library, wrapped within the ELSI *library of solvers*, which offers a uniform interface allowing for the computation and retrieval of the main electronic-structure object (the density matrix) as a function of the Hamitonian and overlap matrices, all without the need of matrix format conversions.
For Leonardo, we deploy an accelerated version of Siesta by simply linking to a version of ELSI (2.9.1) compiled with CUDA. We use the *gcc* software stack, as there is no special need for the offloading features of the *nvhpc* package. For future work we will still need to work around bugs in the *nvfortran* compiler that prevent its understanding of some (legal) constructs in the Siesta code.

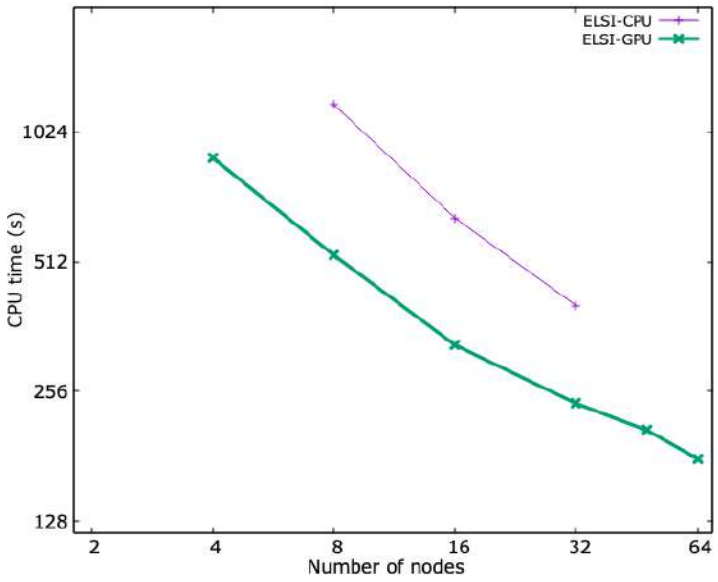

Fig. 8. Execution time versus number of nodes in Leonardo for the solver stage in a single scf step of the covid-12k-diag benchmark with dzp basis set (with a total of approximately 100000 orbitals). The ELSI built-in ELPA solver is used, both with and without GPU acceleration, using 32 MPI ranks in each node.

A special characteristic of Siesta is that it is very efficient for large systems, with a baseline performance that easily beats most other codes in the MAX suite (with the possible exception of BigDFT). For the initial benchmarks in this project we have chosen a system (a piece of the sars-cov-2 virus in a bath of water molecules) with around 12000 atoms. For the largest basis-set used (of dzp quality), that amounts to matrix sizes of the order of 100000. Yet even for such a very large system, the time spent in each computation of the electronic structure (a step in the self-consistency cycle) is of the order of a few minutes on a few tens of nodes.

We present sample benchmark results to help explore finer issues of performance. For the largest system size (Fig. 8), the overall speedup between the GPU-accelerated version and the CPU-only version is around 2.5X.

An alternative way of looking at these kinds of scaling data is to create a "cost vs time-to-solution" plot. In it, perfect scalability would be a horizontal line, and the marginal increase in cost for a given time-to-solution is represented by the (negative) slope of the curve.

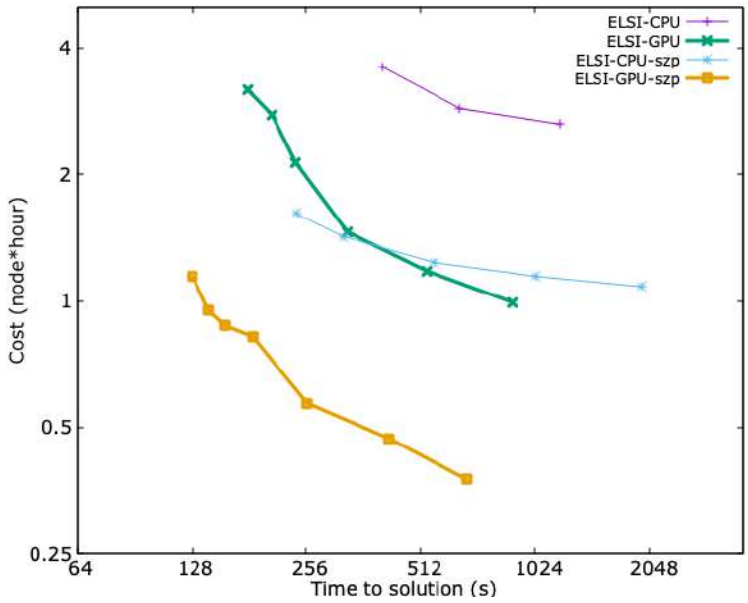

Fig. 9. Cost vs time-to-solution plot for the solver stage in a single scf step of the covid-12k-diag benchmark. Data is presented for both the largest size with dzp basis set and an intermediate size (szp basis, approximately 70000 orbitals). Other details as in the previous figure. The number of nodes used grows from right to left. Deviations from the horizontal indicate reduced scalability.

In Fig. 9 we also include data for an intermediate-size system (with szp basis set and around 70000 orbitals). We can see that, while the time-to-solution is still decreasing for 64 nodes (256 GPUs), the marginal cost is already quite high.

In order to investigate the issue, we performed an initial profiling session via JUBE with Nsight Systems by instrumenting CUDA and MPI, and by filtering our measurement on the solver only with NVTX ranges. Firstly, we observed for the dzp basis set that the memory operations done by the GPU do not include peer to peer copies. That is, GPUdirect is not exploited, and MPI communications stage the buffer through the CPU. This induces expensive copies before and after the MPI communications.

Further, the trace of the largest simulation at 32 nodes revealed that the ratio between time spent moving data and the actual GPU kernel time was 40/60, implying a rather high memory-movement overhead in this setup. We suspected that this might have to do with the granularity of the data movement to-and-fro the GPUs, as 32 MPI ranks per node were used, and thus each GPU serves 8 ranks. Hence, smaller data packets, plus increased latency coming from MPI, could be at the root of the degradation of performance. We therefore ran a more comprehensive profiling session using the newly developed JUBE scripts, to analyse how the gpu memory and kernel time vary with the number of tasks per node. Details are reported in Sec. 5.3.

In summary, we have deployed Siesta in Leonardo, laid the groundwork for benchmark and profiling, and have identified some issues worth exploring in future work.

4.4 BigDFT

In order to permit a more general approach, the BigDFT suite adopted a *remotemanager* workflow orchestrator, which is released as another module of the collection of MAX software libraries. It enables the usage of the remote resources from a jupyter notebook running on a workstation, thereby easing the analysis of the calculations and their results. It has been shown to be compatible with two-factor authentication and for this reason it represents a light-weight alternative to other orchestrators like AiiDA. The notebook associated with this benchmark campaign can be inspected and downloaded at the following URL[11]. The advantage of such an approach is that it can deploy tests and benchmarks on any remotely accessible supercomputer. With this approach, the user can (i) Execute calculations (production, testing and benchmarking) on a remote architecture without having to explicitly deal with intermediate layers of data transport, connection, and scheduling, (ii) seamlessly run jobs different version of the code, for instance, different compilers/libraries, from the same notebook, (iii) re-compile the code in the remote machine, for instance to test nightly code snapshots against stable releases.
Here we can see an example of remote re-compilation of the code on Leonardo thanks to remote manager interoperability (RM)

```
leonardo=RM.get_computer(host='leonardo',flavour='gnu',sshpass_override=sshpass)
prefix=join(base_dir,'build-gnu')
set_prefix(gnu,prefix)
remote_compilation=RM.code_compiler(url=leonardo,
                                    builddir=prefix,
                                    prefix=get_prefix(gnu),
                                    sourcedir=remote_sourcedir,
                                    action="build",
                                    rcfile='/opt/bigdft/sources/rcfiles/leonardo-gnu.rc', #the jhbuild rcfile for compilation
                                    builder='"$SOURCEDIR/Installer.py -y"'
                                   )
```

The BigDFT code was in such a way installed on Leonardo architecture. As Leonardo is a hybrid platform, we have been focusing on testing the suitability of the GPU acceleration with different runtimes. First of all, the code has been compiled with both CUDA and OpenCL acceleration support. As a matter of fact, the code employs two different acceleration strategies which can run efficiently on NVidia GPUs. The wavelets convolutions, which are implemented via dedicated kernels (as a part of the *liborbs* library), are accelerated by OpenCL programming paradigm, whereas the Interpolating Poisson Solver, employed for the calculation of exact-exchange functionals, has been accelerated with the usage of CUDA CuFFT. The latter acceleration is also part of a dedicated library of bigdft-suite (*PSolver*), which is a part of the MAX software suite.

[11] https://github.com/BigDFT-group/resources/blob/main/technical_note/CPUGPUBenchmarking.ipynb

The code can then be executed on the machine by triggering the different accelerations. Once again, this approach can be performed in a easy way thanks to the RM module:

```
leonardo=RM.get_computer(host='leonardo',flavour='gnu',environment='1.9.4-gnu',sshpass_override=sshpass)
prefix=join(base_dir,'build-gnu')
from benchmark import run_bigdft
ds=Dataset(run_bigdft,local_dir='leonardo-2CzPN-gnu',
           dbfile='leonardo-2CzPN-gnu.yaml',
           remote_dir=remote_gnu,
           name='2CzPN',url=leonardo, time='02:00:00',skip=False)
# we first pile up the CPU calculations
for run in bench:
    ds.append_run(lazy=True,**run.copy())
# then we trigger the accereration
for run in bench:
    inp=Inputfile(run['arguments']['input'])
    inp.use_gpu_acceleration(flavour='OCL') #and/or flavour='CUDA'
    run['arguments']['input'] = dict(inp)
    run['gpu'] = 4 # per node
    run['name'] += '-ocl'
    ds.append_run(lazy=True,**run.copy())
```

This created a benchmarking campaign producing all the data which can then be transported back on the local workstation for analysis (vide infra). This approach is completely general and can be extended to essentially any remote architecture, thanks to its compatibility with 2FA.
An automatic conversion of this deployment method into the JUBE approach is under implementation at the time of writing this report. In the following section we analyse and comment on the results obtained.

4.5 Fleur

For the FLEUR code a list of benchmarks using the JUBE framework has been created and can be found in the JUBE4MAX repository[12]. These cover a significant part of the features of FLEUR and also cover different system sizes. As the workloads are separated from the setting of the computational parameters in the JUBE scripts this allows for an easy and systematic study of the performance of the code in typical simulation scenarios. However, a strong focus lies on the most difficult to scale eigenvalue parallelization and only a single test uses more than a single k-point. While this is useful for the performance optimization of the code and for the ultimate scalability, one should keep in mind that in particular for small systems the k-point parallelisation adds an additional level of parallelism with nearly perfect scalability.

[12] https://gitlab.com/max-centre/JUBE4MaX/-/tree/fleur

On Leonardo we compile FLEUR using the Nvhpc compiler suite. This compiler supports the OpenACC GPU programming used in the code and also provides some tuned libraries like the cuBLAS/cuSolver packages. Our configuration procedure will then compile the ELSI library as well as the HDF5 library in place to achieve maximal compatibility.

5 Show cases of benchmarks and characterization towards Exascale

In this paragraph we present early results of performance assessment done on Leonardo, mostly by using the ecosystem of JUBE scripts described above. The analysis embraces important topics towards exascale on heterogeneous architectures, such as the efficiency of GPU to GPU communication protocol, the implementation of workflows to avoid synchronisation penalties and the interoperability between different programming models on GPUs to ensure maintainability together with performance.

5.1 Band parallelization for Gamma cases in QuantumESPRESSO

For calculations using one k-point, so-called *Gamma calculations*, the only viable parallelization for strong scaling is the band group parallelization, where operations on the wave functions are distributed among many band groups. This parallelism will be also instrumental for keeping the size of R&G groups optimal, and speeding up the calculation by scaling up the number of band groups. In this showcase, we present the characterization and profiling of the current implementation of band group parallelism; the workload is the one of the CNT10POR8[13] benchmark, executed on Leonardo. Such implementation - being mainly tailored as supplementary to R&G for homogenous parallel machines - is clearly not ideal for GPU-based ones.
Figure 10 shows the time to solution obtained by running the develop version of the code with 24 GPUs without band group distribution, to 48 GPUs (2 band groups) and 96 (4 band groups).

[13] The system simulates a carbon nanotube functionalized with a porphine; contains more than 1500 atoms and works with more than 3000 bands. Input is available here:
https://gitlab.com/max-centre/JUBE4MaX/-/tree/main/max-inputs/workloads/qe/pw/

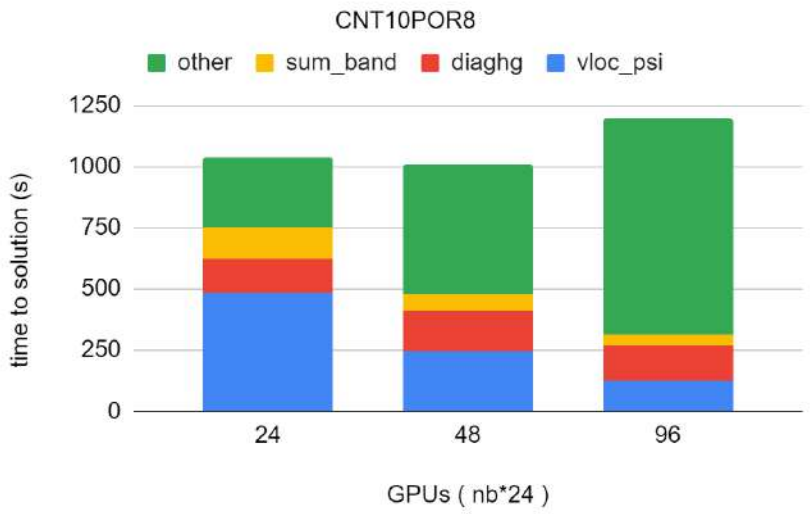

Fig. 10. Time to solution and some of the most relevant contributions of PWscf to simulate 32 steps of CNT10POR8 ground state calculation. Beyond 24 processes of the R&G kind, band distribution is used. The symbol nb refers to the number of band groups.

By looking at the relative time spent in some of the most time consuming routines, we clearly observe that the effectiveness of band distribution is limited to some kernel only, and in particular to vloc_psi, the main driver for FTs.

To obtain more insight concerning the overheads and benefits of the band group parallelization, we profiled the simulations with Score-P by instrumenting the CPU only[14]. Figure 11 reports the contributions in percentage to the total time, i.e. the sum of the time spent by each MPI task from the start to the end of the program.

[14] In this measurement, the GPUs execution time is not included. The time labelled with "OpenACC" and "CUDA" refers to the time spent by the CPU in the OpenACC runtime and CUDA API.

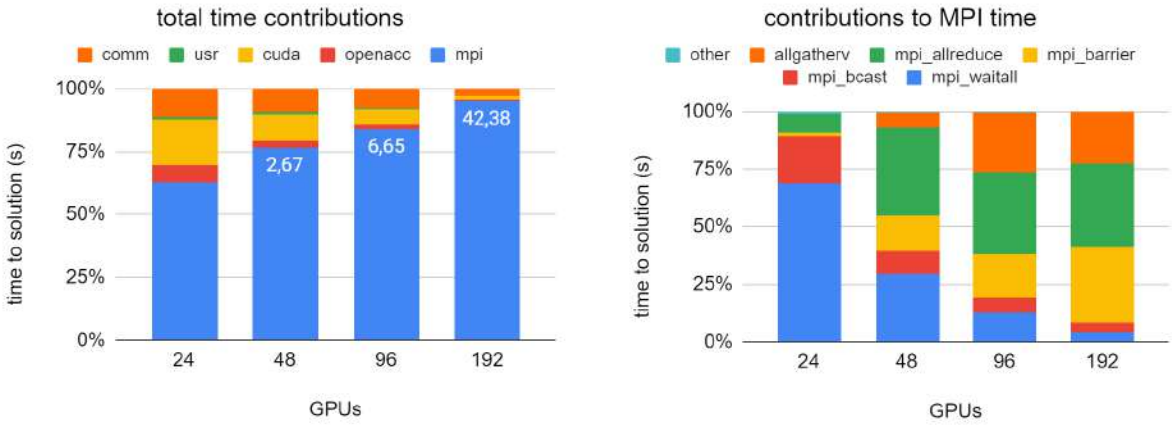

Fig. 11. (Left) contributions to the total time on the CPU measured with Score-P. The code is not instrumented for GPU events. (Right) contributions to the MPI time measured with Score-P. Note the decrease of MPI_Waitall and the increase of MPI_Allreduce and MPI_Allgatherv with the band index.

The characterization in the left panel of Fig. 11 indicates that most of the time on the CPU is spent in MPI calls; the numbers in the blue columns show the relative increase of time spent in MPI with respect to the run without band parallelism (24 GPUs). The right panel of Fig. 11 highlights which MPI routines limit mostly the scalability, when the number of band groups is varied. While the percentage of time spent in MPI_Waitall - the main bottleneck for scaling with R&G[15] - decreases, new communications of the MPI_Allgatherv kind arise, together with an increase of the time spent in MPI_Allreduce.

Band group processes are indeed at a higher level in the MPI hierarchy with respect to R&G ones MPI_Waitall are intra-band communications triggered by the distributed driver for FFTXlib on GPUs, and thus these are not directly affected by band distribution. On the contrary, other communications are needed to broadcast the results of h_psi and s_psi to all band groups (this currently occurs inside the global wrapper of these kernels) and during other global operations such as Gram-Schmidt orthogonalization or the evaluation of matrix elements. These communications are executed either with MPI_allgatherv calls, or in other cases with even less efficient MPI_Allreduce calls. This is clarified by the snapshots of the profiling report at 96 GPUS (24 nodes with 4 band groups and 24 R&G each) visualised from CUBE[16] (Fig. 12), where the location of the most time consuming MPI calls of the Allreduce and Allgatherv kind is highlighted.

The reasons for these large communication overheads, that make current implementation of band group parallelism inefficient, are: (i) the large memory footprint, which forces the use of too many R&G ranks

[15] More details on R&G scaling are reported in Sec. 6.
[16] https://www.vi-hps.org/projects/score-p/

within each band group; (ii) the use of collective MPI reductions for exchanging data among many nodes; iii) the significant work unbalance when performing collective calculations among all band groups.

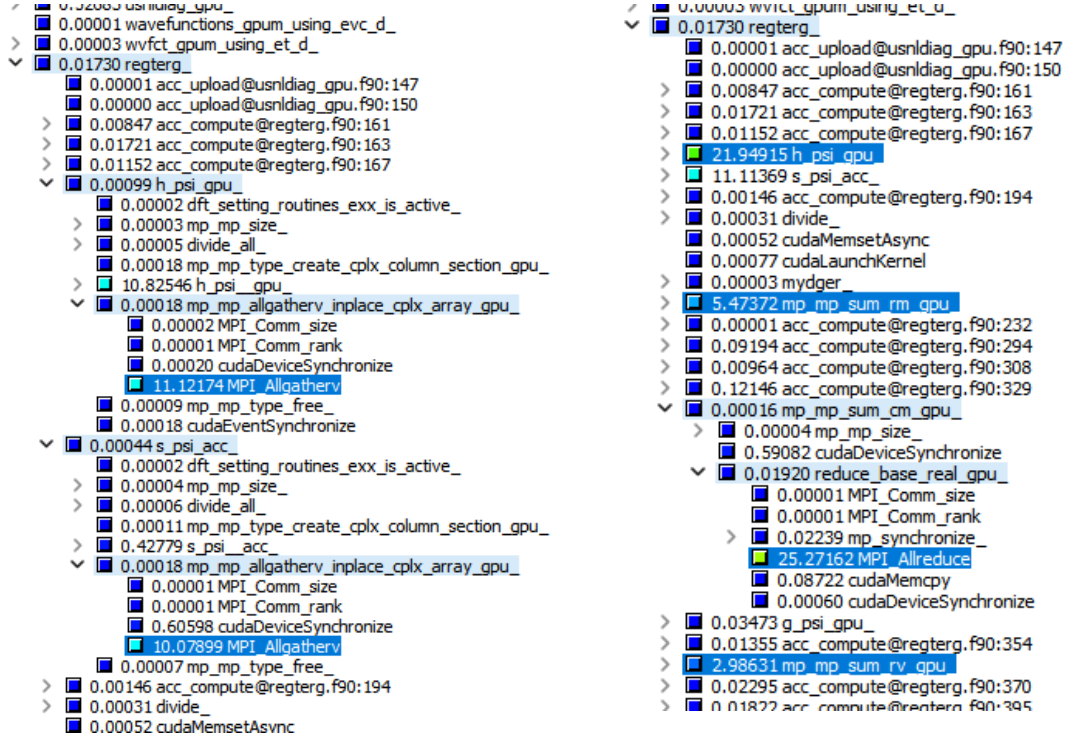

Fig. 12. Profiling summaries visualised from CUBE and obtained with Score-P by instrumenting the CPU only at 96 GPUs (24 nodes, nb = 4). The panel on the left shows the percentage of time spent in MPI_Allgatherv from the different calling paths; the same information is provided for MPI_Allreduce in the right panel.

This indicates the main refactoring actions needed for the band-group parallelization:

1. The wavefunction data should be effectively distributed, each band group allocating only the data for its assigned wave-functions. This will reduce the memory footprint on GPUs, the number of R&G ranks of each band group, the volume of inter-group exchanged data, consequently mitigating the communication overhead between band groups.
2. The data-exchange between band groups that currently occurs with MPI_Allreduce calls of the common large buffer allocating all wave-functions must be refactored, avoiding collective MPI calls and resorting to asynchronous point-to-point communications, optimising the overlap between data-exchange and computation.
3. Calculations requiring data exchange with other band-groups (e.g calculation of $\langle\psi_i|H|\psi_j\rangle$ matrix elements or Gram-Schmidt orthogonalization) should be reorganised so as to maximise their concurrent execution and/or their throughput.

5.2 OpenCL and CUDA accelerations (BigDFT runs)

We have chosen a set of systems made of replicas of organic molecules (2CzPN), which are employed in the study of organic photovoltaic materials. Such a test case has been chosen in this way for various reasons. First of all, it is a set of systems for which it is potentially interesting to study the influence of the exchange and correlation approximation, and therefore it can be run in production both with traditional semilocal functionals (PBE) and hybrid functionals (PBE0). Secondly, we have chosen this system as representative of a production environment, in the sense that the workload is not designed to be well-balanced among the different CPU cores. Each of the 2CzPN molecules has 83 Kohn-Sham orbitals, and for this reason it is interesting to observe how the architecture behaves even when the computational workload is not perfectly balanced (we recall that Leonardo has 32 CPU cores per node, and 4 GPU). As explained in the previous chapter, these tests have been run with the *remotemanager* workflow orchestrator, which is released as another module of the collection of MAX software libraries.
The first set of calculations that has been performed is devoted to the inspection of the effectiveness of the OpenCL acceleration. The OpenCL convolutions have been inserted in the code well before the advent of the A100 platforms (the first version of the OpenCL accelerated BigDFT code dates back to 2011). It is therefore important to understand where we stand with a codebase that has not been tailored to the specific architecture. Results are shown in Fig. 13. Runs of various sizes (1, 2 or 4 molecules, of 54 atoms each), have been performed with a variable number of nodes, with different MPI-OMP repartitions (indicated in the tick labels of abscissas). Such runs have been analysed in terms of their contribution to the different categories during runtime. The time spent in the wavefunction optimization is separated in communications, linear algebra, and in the specialised kernels employed for the convolutions (OpenCL) or the FFT (which in the case of PBE0 can be accelerated thanks to CuFFT).
In the case of a PBE calculation, where only the convolutions can be accelerated, we can see that the code can reach a speedup factor of about 3, depending on the MPI-OMP repartition and the number of nodes employed. This is remarkable information, considering that the codepath related to OpenCL is not specified to the A100 architecture and that the convolutions are not the exclusive operation performed - yet representing a significant fraction of the overall compute time. For each node, it can be seen that it is nonetheless important to keep the number of OMP threads per MPI on a number of at least 4, to reduce the overall time spent in the communications (as the overall number of MPI tasks would be thereby reduced).

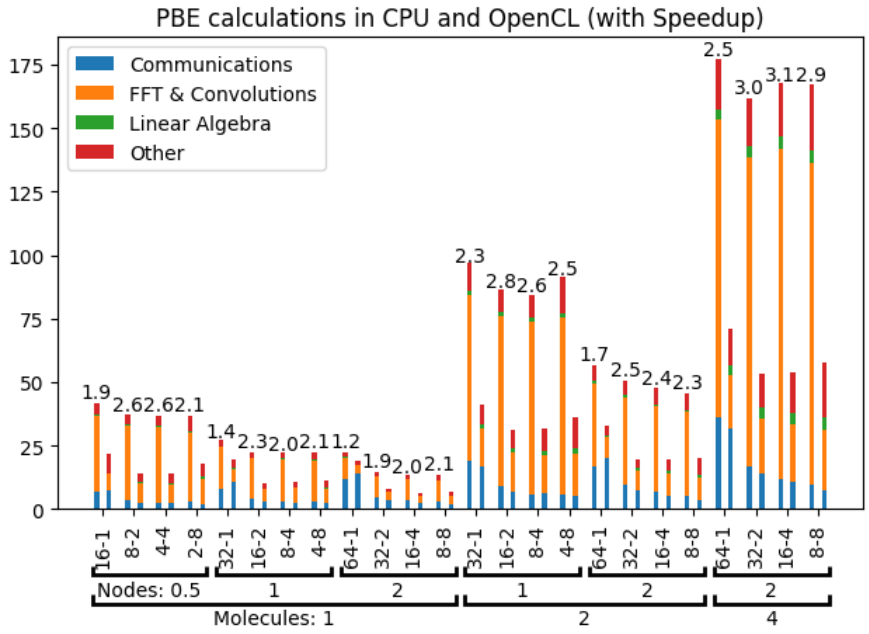

Fig. 13. Walltime required for the full SCF optimization of different sets of 2CzPn molecules with different repartition of nodes and MPI-OMP tasks-threads, respectively. The runs are ordered by the number of molecules, the amount of nodes employed and the number of OMP threads per MPI task. The tick in the x axis indicates the MPI-OMP configuration, while the underbraces regroups nodes and molecules. For each repartition, the bars on the left are related to a homogeneous pure CPU runs, whereas the bar at the right indicate the speedup related to the OpenCL acceleration, which is visible in the FFT & Convolution category (which is the one that is accelerated). The numbers on the top of the CPU bars indicate the corresponding speedup that is obtained thanks to this acceleration.

We have also performed the same approach with a PBE0 calculation, which once again is enabled by the usage of the PSolver library, which can be accelerated with the usage of CuFFT. The overall behaviour is illustrated by Fig. 14, where we can see that the time spent in the FFT grows considerably in the CPU case, which is a well-known behaviour of exact exchange calculations. Still, the CuFFT acceleration, together with the ability of the PSolver library to overlap communications and calculations - and the usage of GPUdirect layer, enables us to reach speedups on the calculations which are larger than one order of magnitude on each of the tested configurations. This is an interesting showcase of a realistic production calculation where the user can seamlessly benefit from the GPU architecture, and the corresponding GPU-aware MPI layer, such as calculations can be enabled which would otherwise take prohibitive computational cost. In Fig. 14 the PBE0 calculations with 2 molecules have been only calculated with GPU acceleration, due to the otherwise large walltime required by the CPU calculations.

All the data represented here are in seconds, and correspond to the overall SCF convergence of a KS DFT calculation.

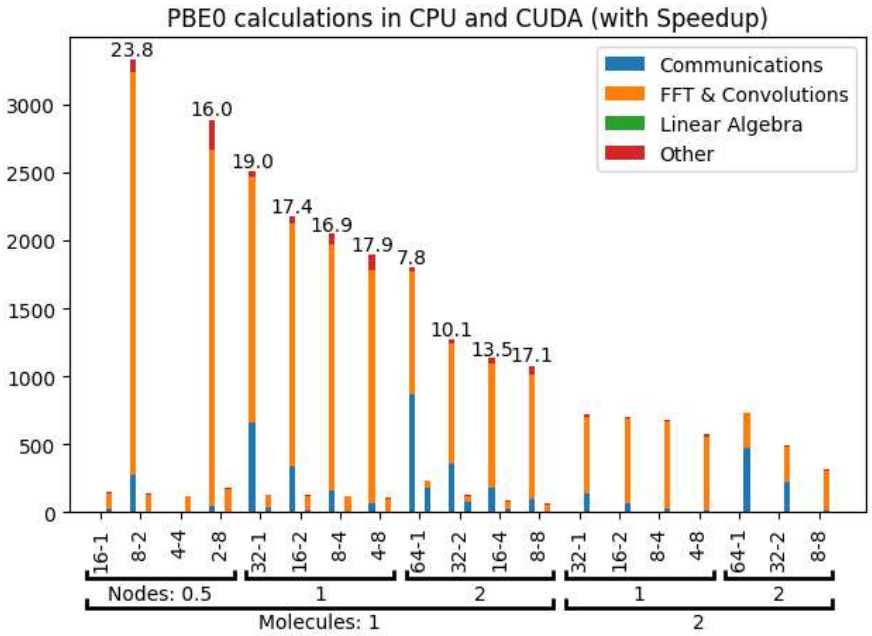

Fig. 14. Walltime related to the acceleration of a set of different 2CzPN molecules with PBE0 Hybrid Functional and CUDA acceleration. Conventions of the plot follow the same guidelines of Fig. 13, with the difference of the fact that the accelerated version of the code employs CUDA for the Exact Exchange calculations. Runs with 2 molecules have been only executed with GPU, in order to keep the benchmark time reasonably limited.

The PBE0 behaviour seems therefore quite interesting to analyse, especially in terms of its bandwidth. Being a FFT calculation, we know that this approach is memory-bound, and it would be nice to understand the impact that the usage of high bandwidth memory would have on the application. In order to do this, we have analysed the behaviour of the Fock mini-app, which reproduces the execution of one single application of the Exact-Exchange Fock operator, on different working conditions. We have compared a workload with 128 KS orbitals on a "small" FFT grid (96 grid points per direction), with a workload of a "large" FFT (256^3 grid points) with 96 orbitals. For reference, the above shown calculations have a grid of large size (203x185x179 plus zero padding). Such a mini app comparison is therefore handy as it enables one to understand the behaviour of the code without having to run the entire SCF cycle. We have performed the calculations on a system consisting of two 4th Generation Intel Xeon Scalable Processors (Intel Xeon Platinum 8480) and four Intel Data Center GPUs Max 1550 which

are interconnected with Xe-Links. Each node in the cluster includes 512GB of DDR5 random-access memory (RAM) and the nodes are interconnected with an InfiniBand HDR fabric.
From the calculations performed it is evident that the smaller workload (96^3) fits into L3 cache and thus induces minimal HBM traffic. The larger workload, in contrast, does not fit in L3 and induces heavy HBM traffic. In the case of the larger workload, the code utilises on average 783 GB/s of HBM bandwidth, which represents approximately 48% of the theoretical peak bandwidth of a single stack. The complex-to-complex FFT with the maximal averaged bandwidth out of all complex-to-complex FFTs achieved a bandwidth of 938 GB/s. Such preliminary analysis, performed on an architecture different from Leonardo, enables us to understand the limitations that the application can have, depending on the runtime regime. Runtime comparisons between the Intel CPU calculations and the CUDA behaviour suggests that the rationale is similar also on NVidia architecture. Detailed measurement on the bandwidth employed on the Leonardo nodes can be performed in the following of the campaign.

5.3 Profiling of solver stage in Siesta

By relying on an external library (ELSI-ELPA) to compute the density matrix, SIESTA inherits all the optimizations for the underlying hardware from the library itself. Our concern in this framework focuses then on identifying the best configuration of resources on Leonardo to make the best use of the external accelerated engine. To shed light on the outcomes of the benchmarks in Sec.4.3, we ran a more complete profiling session of the solver stage via JUBE, with a number of tasks per node ranging from 32 to 4, the latter matching MPI ranks to number of GPUs in each node. Starting from the smaller size system, the profiling data for mpi rank 0 show that the ratio between time spent in computing (kernel operation) and implementing data movement of any kind is improved with one MPI rank per GPU, as reported in Fig. 15. This is likely due to the fact that in the initial configuration (32 tasks per node) MPI distribution reduces the size of the data processed on the GPU, up to the point where kernels are too small to be executed efficiently on the device with respect to the time spent moving data. Moreover, if there is not enough work for the CPU between one GPU operation and the other, scheduling multiple GPU operations at almost the same time might result in large latencies between the CPU launches and their actual execution on the GPU.
While binding a smaller number of tasks per device might improve the GPU usage, we should consider that the solver stage is composed of several phases, and that each phase might execute kernels with different workloads on the GPU. Further, the first step of each scf cycle implements Cholesky decomposition, which is not offloaded. To understand the interplay between these parts, we characterise their time to solution for different numbers of tasks per node by retrieving this information from the application logfile via JUBE (Fig. 15).
By looking at the trend of the Cholesky step, we observe that the time to solution decreases almost monotonously with the number of tasks. This can be ascribed to the fact that Cholesky uses the CPU, and in this range of node numbers it still benefits from workload distribution via MPI tasking.

Differently from Cholesky, the time to solution of the offloaded phases oscillates visibly with the number of tasks per node (see in particular solve_std and to_std in Figs.15 right panels). In particular, the solution to the standard problem (solve_std) is the most time consuming and runs more efficiently with one MPI task per GPU. solve_std is still scaling from (4, 4) to (16, 4). Using more nodes, e.g. (32, 4) would not provide a significant speedup, and would also deteriorate the efficiency of the transformation to the standard problem (to_std).

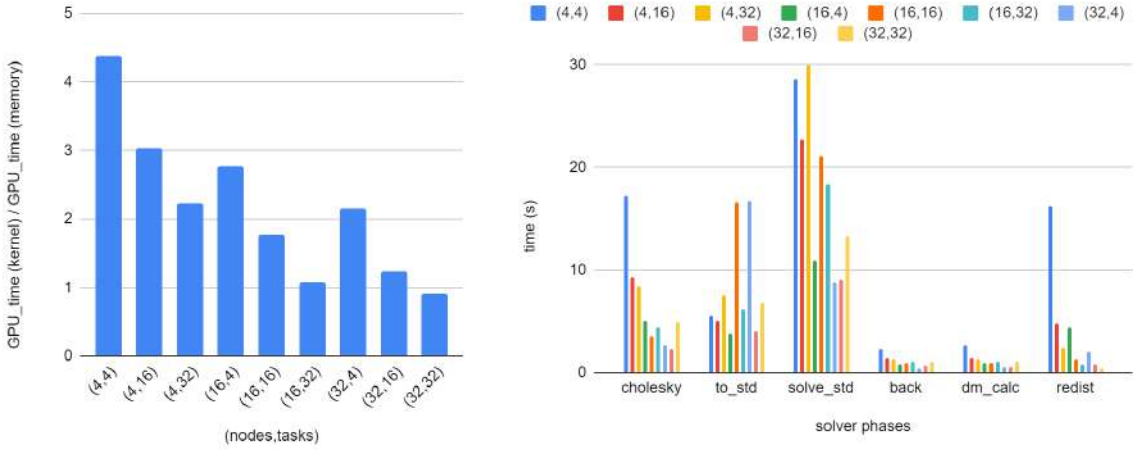

Fig. 15. (Left) relative fractions of time spent in kernel over memory operations on the GPUs and (Right) execution times of different solver stages. The workload is the one for the smallest benchmark, with a sz basis set (about 30000 orbitals). In this benchmark we analyse the performance of different numbers of mpi tasks per node, that are (4,16,32).

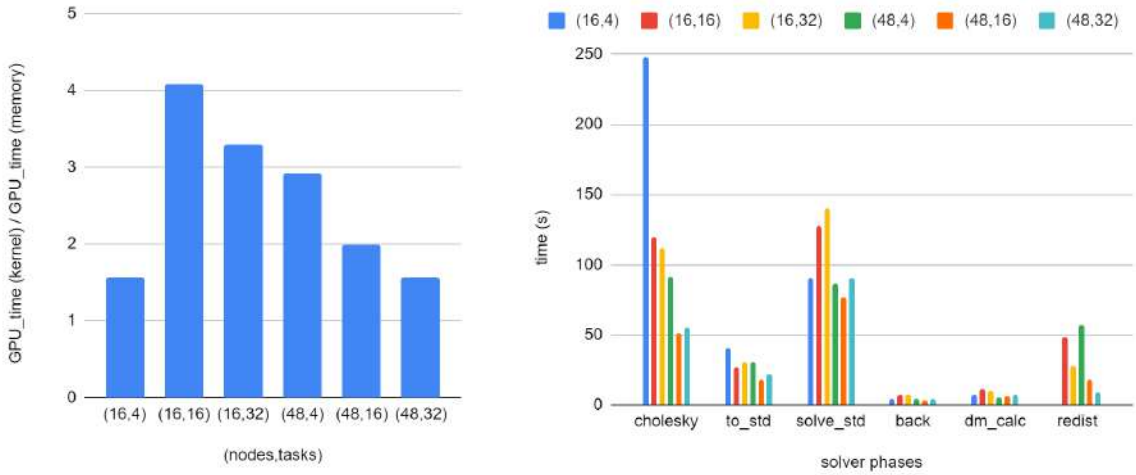

Fig. 16. (Left) relative fractions of time spent in kernel over memory operations on the GPUs and (right)Execution times of different solver stages. The workload is the one of the largest benchmark, with a dzp basis set (about 100000 orbitals). In this benchmark we analyse the performance of different numbers of mpi tasks per node, that are (4,16,32).

This conflict is leveraged by increasing the size of the workload, as visible from Fig. 16, where however the best performances are achieved by using 16 tasks per node rather than 4.
A better understanding of this behaviour with increasing system size requires increasing the granularity of the measurements; however, this early analysis shows the difficulties in identifying the best set up of resources to scale the application efficiently. At present, any further improvement in the time to solution observed for solve_std phase is potentially reduced by the time spent in Cholesky decomposition. This step is being targeted for proper GPU offloading by the ELPA developers, and an implementation is expected very soon. The results obtained for the largest system suggest to test the activation of Multi Process Service[17], in order to improve the scheduling of operations on the accelerator when more tasks offload kernel and memory operations to the same GPU. Further, as long as some parts of SIESTA benefit from a different number of tasks per node, we will consider, in collaboration with WP1, the implementation of a subgroup of MPI ranks to be used for those parts where GPU operations are involved.

5.4 Avoiding load imbalance on response function calculations in Yambo

Section 4.2 shows two strong scaling tests of the Yambo code simulating the same system but with different ground-state parameters. In both cases the main driver that takes the most of the computation time is the one calculating the irreducible response function (*Xo*). This driver allows for a parallelization strategy distributed across five MPI levels: transferred momenta (q), space variables (g), momenta (k), conduction bands (c), valence bands (v). In both the test cases, we studied the scalability by distributing the processes in large part on the conduction bands level. In particular, we decided to disable the parallelization level of the transferred momenta despite it being an embarrassingly parallel level. In fact, when enabling the q level a load imbalance among the processes is encountered. This happens because the time spent on every q point can be very different depending on the number of crystal symmetry operations associated with it. An example is shown in the table below, where the times needed for the computation of the Xo(q) fragments are reported for the small GrCo system (see section 4.2, Fig. 7 top frame) running on 5 nodes of Leonardo. The first q point corresponds to the $\mathbf{\Gamma}$ point (the most symmetric), and the last to the high-symmetry $\mathbf{K}$ point. As expected, their computation is faster.
In the Yambo input file it is possible to set a variable (whose name depends on the approximation used for the frequency dependence of Xo) defining a range of q points Xo has to be calculated for. This is possible because the transferred momenta are completely independent and a different database, based

[17] https://docs.nvidia.com/deploy/mps/index.html

on netCDF output file, is generated for every q point. Thus, it is possible to run every q point as an individual job, and at the end of the calculations all the databases are available in the same output directory of the run. Then, a final run reads all the previously calculated databases and concludes the GW workflow.

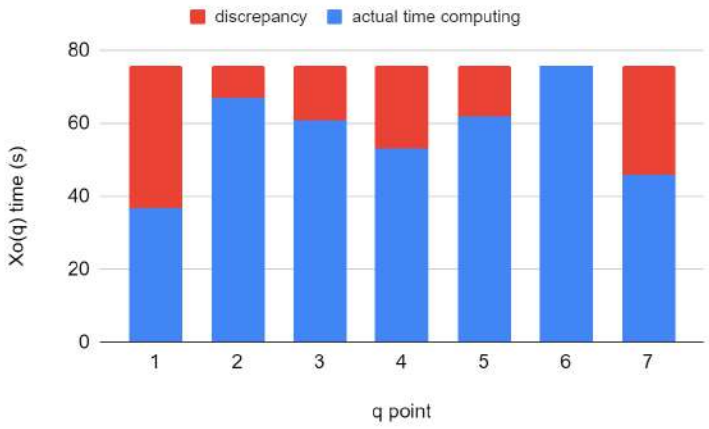

Fig. 17. Time to compute Xo for a given q point (blue column)
and discrepancy with respect to the largest time (red section of the column).

If the number of q points is large, it becomes difficult to manually manage this feature of the Yambo code. A workflow manager would enable launching separate simulations for each q point with tailored resources and avoid the penalty of synchronisation barriers. Moreover, it would simplify the implementation of dynamic scheduling, to balance the workload among processes. Interestingly, such a workflow could be easily implemented in the JUBE framework by exploiting the "step dependency" feature of the tool. However our implementation does not allow for checking unfinished steps (although it is possible to resubmit crashed runs). A workflow manager like AiiDA[18] could be instead the best solution, thanks to the fault tolerance feature of the YamboRestart workchain that is available in the aiida-yambo plugin. In fact, YAMBO implements a typical high-throughput scenario, which suggest to investigate its integration into a workflow manager to foster scaling at the largest scale

6 Proof-of-concepts for GPU exploitation: GPUdirect communications in FFTXlib

The acceleration provided by GPUs in heterogeneous programs can be jeopardised by expensive data movements between the host and the device. This bottleneck can be even more detrimental in

[18] https://www.aiida.net/

MPI-distributed programs, where the data is staged through the CPU before and after the MPI communications. To overcome this bottleneck, vendors provide GPU-aware protocols to exchange data via MPI, which bypass the copy from/to the host by moving them directly among the accelerators (GPUdirect). FFTXlib, the low-level library used by QE for serial and distributed 3D Fourier Transforms (FTs), contains drivers tailored for different architectures, thus it can be used as a proof of concept to investigate the effectiveness of peer to peer (P2P) and RDMA GPU-to-GPU communications on a distributed network.

FTs operations are highly time consuming and widely used in codes like QE, because they allow for handling matrix algebra operations by shifting them to the space where they become less computationally demanding. Thus, the wave function or the electronic density of the system are repeatedly Fourier transformed and anti-transformed in order to switch from the real space representation to the reciprocal space one, or vice versa. FFTXlib performs and optimises these operations for various serial and distributed hardwares. In the slab decomposition, xy-planes are distributed along z-direction to MPI tasks, while the spherical reciprocal space is divided into z-sticks covering the whole xy-circle.

The standard sizes adopted throughout the years are 16 bands for each batch and 4 bands for each sub-batch, except for the remainder. GPUs and HPC technology in general, however, have significantly evolved since the first implementation of the batched FFTs, and the average size of each run has grown accordingly. Therefore, a revaluation of the optimal set of parameters depending on the size of the physical systems and on the available HPC resources is expected to assess such values.

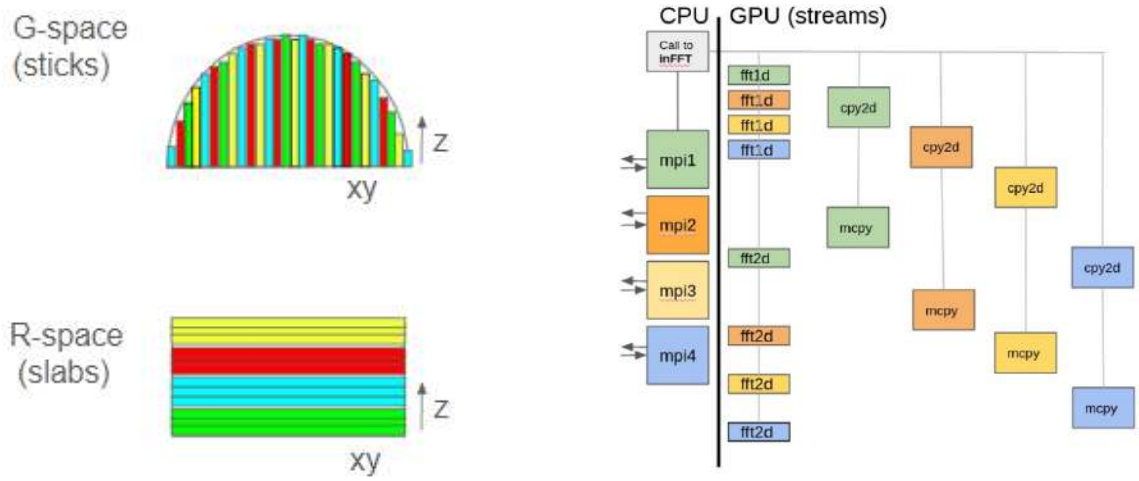

Fig. 18. (Left) Scheme of distribution adopted with R&G parallelization and slab decomposition. The reciprocal space is divided as sticks in the xy direction, while the real space is distributed as planes along z. (Right) GPU driver for the FTs, from the reciprocal to the real space. The steps can be summarised as follows: (i) compute one-dimensional FFTs along z-axis on

the GPU; (ii) download the data from GPU to CPU; (iii) distribute domain xy-slabs by moving data between CPUs; (iv) upload the data to the GPU; (v) two-dimensional FFTs over xy planes on the GPU.

With this purpose, we will analyse the effect of batching, subbatching and GPUdirect on the time to solution of the whole simulation (Fig. 19 left panel) and `vloc_psi` (Fig. 19 right panel), the routine that relies the most on FFTXlib. We started considering the smallest test case, AUSURF, to assess mainly intra-node performances. In the following table we provide the parameters of the 5 configurations under study.

many_fft	**sub-batch size**	**async**	**GPUdirect**	**case**
1	-	-		A
16	16	False	no	B
16	16	True	no	C
16	4	True	no	D
16	4	True	yes	E

A : cpu-like algorithm (bands are treated sequentially)
B : multiple batches at once. The Fourier transforms (1z and 2xy) are still sequential with respect to the band index, but the psi for multiple bands is loaded to/from the GPU and between GPUs in batches. The size is given by many_fft. This tests helps to understand the effectiveness of batching, without accounting for asynchronous cpu-gpu execution (sync)
C : multiple batches at once. The Fourier transforms (1z and 2xy) are still sequential with respect to the band index, but the psi for multiple bands is loaded to/from the GPU and between GPUs in batches. The size is given by many_fft. MPI can overlap with data movement and GPU computation (but no overlap between GPU computation and data movement).
D : effect of subbatching. The batches are divided into sub-batches, there might be overlap between MPI, GPU computation and data movement. There can be overlap only between GPU computation for batch n+1 and data movement for kernel n. No GPUdirect.
E : GPUdirect, i.e. there is no data movement between host and the device. The overlap is between kernel computation and GPU to GPU communications.

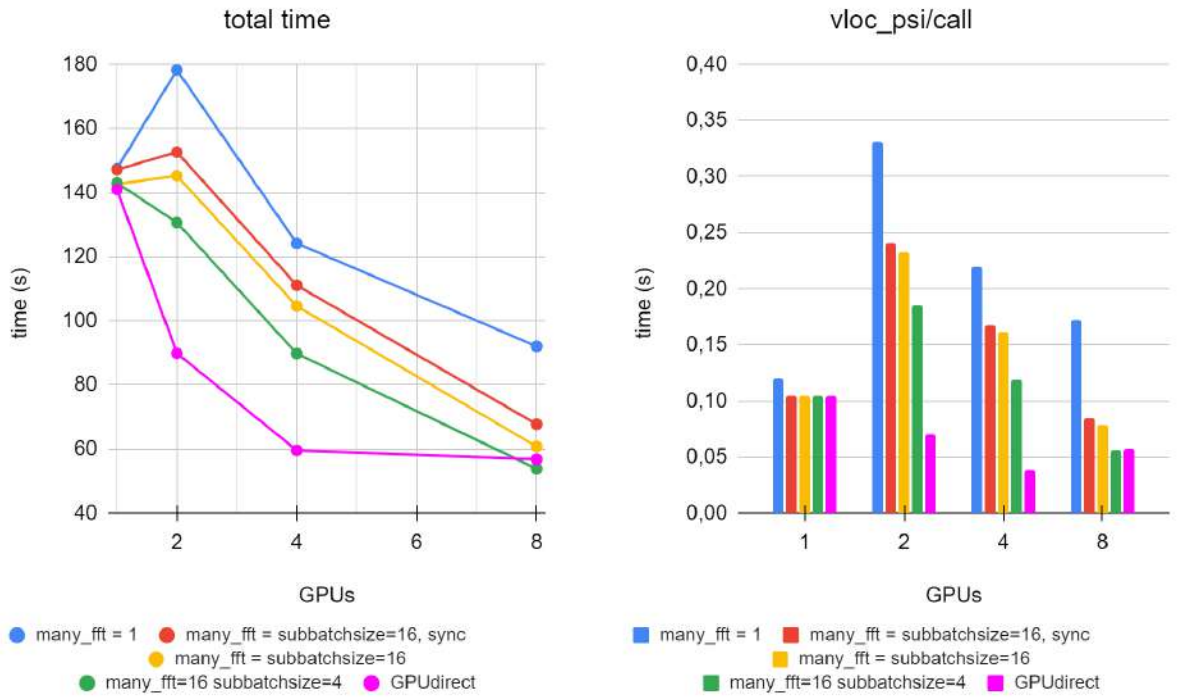

Fig. 19. Time to solution of the (left) whole simulation and (right) vloc_psi for the different set up as described in the Table above.

The reason for the limited efficiency beyond the intra-node with GPUdirect communications deserves further investigation, which will be pursued in the next months, possibly with the use of a suitable FFTXlib miniapp in order to reach more general conclusions, not being constrained by the overall size of the external workload. However, we started considering network optimization such as tailoring the RENDEZ-VOUS protocol for non-blocking communications, which leads to a speed up of the time to solution for the aware case of about 1.5. Preliminary results are reported in Fig. 20 on the right panel.

In addition to customising the use of the given network, we tested the performance of the same workload on the LUMI cluster. While the status of the QE version ported with OpenMP offload is generally not as mature as the NVIDIA version, the OpenMP offloaded version of the vloc_psi routine from PW is quite the same. That is, batching and subbatching have been implemented also for the AMD technology with the same pattern adopted for the historical GPU backend. The main difference is the programming model, OpenACC with CUDA (cuFFT for the 1D and 2D FFTs on the GPU) for the latter, and OpenMP 5 plus HIP (hipFFT for the 1D and 2D FFTs on the GPU) for the former.

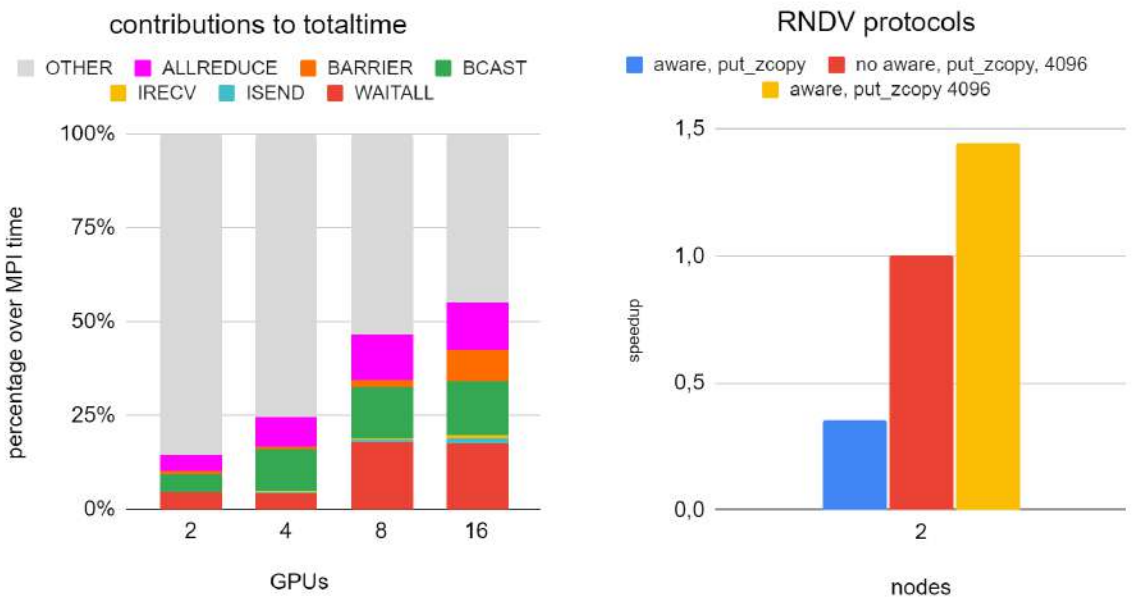

Fig. 20. (left) contributions from MPI and other events to the total time on the CPU for AUSURF, aware version, according to Score-P measurements; (right) speedup of the time spent in vloc_psi by varying the rendez-vous protocol and by setting the UCX_NET_DEVICES to the nearest NIC.

The use of a different programming model has determined relatively small differences between the two implementations of vloc_psi. In particular, the unavailability of launching an OpenMP5 kernel to a specific stream (which is instead possible with OpenACC/CUF) required to replace some normalisation operations at the end of FFT computations with HIP kernels. This is a small part of the whole calculation of FFTXlib, which should not remarkably affect the overall performance.

The comparison between the performances of the two versions must take into account different factors, in particular the differences in the hardware and accelerator topology of the clusters for deployment. Leonardo is indeed equipped with 4 NVIDIA GPUs per node, while LUMI has 8 AMD Graphic Computing Dies distributed on 4 GPU modules per node. Together with a different topology, LUMI has a network based on an Infinity Fabric interface, which provides different BWs according to the distance between the GDCs (from 400 GB/s to 100 GB/s)[19]. First, we observe that the overlap between communication and computation coming from subbatching efficiently reduces the time to solution also in this version. However, differently from the OpenACC version on Leonardo, Fig. 21 shows that the GPU-aware install is the most efficient both on a single and two nodes with respect to the non-aware version. Interestingly,

[19] https://docs.lumi-supercomputer.eu/hardware/lumig/

the trend of the time to solution for both aware and non aware communications is the same in terms of node occupancy.

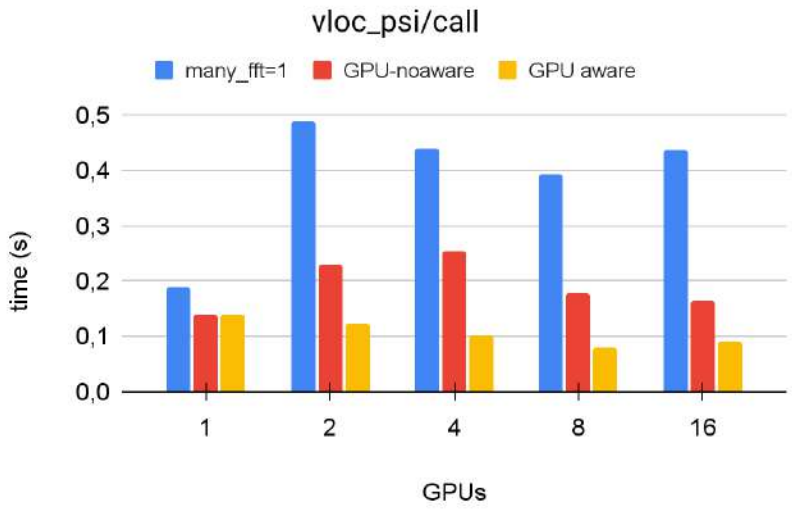

Fig. 21. Time to solution of vloc_psi/call on LUMI for three different configurations using many_fft=1 (CPU-like driver), with and without GPU awareness.

The reason for the different speed up between the two versions deserves further investigation, in order to understand if it should be ascribed to the accelerated FT kernels, the memory copies or the network topology. This aspect, together with the limited efficiency of GPU-aware `vloc_psi` already addressed in Sec. for heavier workloads, will be the object of a deeper investigation in the next months, following the objectives of T3.2.

7 Deployment of the lighthouse applications on the EuroHPC systems

7.1 Vega

The EuroHPC Vega system, as the first available EuroHPC system, has been extensively used for benchmarking and development calls by MAX code developers already before the project started. We have seen all the codes deployed and tested.

HPC Vega is an Atos/Eviden BullSequana XH200 system with published sustained performance of 6,8 petaflops and peak performance of 10,1 petaflops. It has two modules, a CPU partition with 960 nodes configured with 256GB memory/node (20% 512 GB) and HDR100 interconnect and a GPU partition with 60 nodes with HDR200. Its total counts are 1920 CPUs (AMD Epyc 7H12, total of 122.800 cores) and 240

Nvidia A100 GPU units. HPC Vega has a high-performance NVMe Lustre fast storage (1PB) and a large-capacity CEPH storage system (23PB). It has been configured as a general purpose system with traditional Computational, AI, Big Data/HPDA, Large-scale data processing with wide bandwidth for data transfers to other national and international computing centres (up to 500 Gbit/s) with the design goal of data processing throughput 400 GB/s from high-performance storage and 200 Gb/s from large capacity storage.

Vega has supported a number of activities involving testing deployments, including workshops and demonstrations. Our MAX team has deployed MAX software also as end-user accessible installed modules: QuantumEPRESSO, YAMBO and SIESTA are available with current versions and documentation. (The same packages are also available on other national systems in Slovenia.)

HPC Vega participated in a testing of automated CI/CD deployments using different dedicated GitLab repositories. We tested the methodology using QuantumESPRESSO as an example application. Deployment included running development repository sync, build, test and benchmark jobs of optimised stable versions of QuantumESPRESSO. Suitability of different methods of repository pulling and triggering of CI/CD pipelines was tested and demonstrated. Different deployments of Git Runner solutions were tested on IJS's NSC, Arnes's HPC and HPC Vega machines in order to evaluate the use of GitRunner ssh executors as well as Jacamar using SLURM executor. Direct technical access with the ability to run GitLab runners on-site, as well as full administrator access to the sites and support at these sites facilitated the procedure.

7.2 MareNostrum-5

Following all the work activities initiated in Leonardo, the pre-exascale EuroHPC MareNostrum-5 supercomputer will soon become accessible to all MAX code developers. Deployment, profiling and benchmarking activities will be carried out as soon as the machine gets into the production stage, which will most likely happen by early 2024.

The machine is supplied by Bull SAS, combining Bull Sequana XH3000 and Lenovo ThinkSystem architectures. MareNostrum-5 consists of two main partitions with different technical characteristics, with a total peak performance of 314PFlops, and their next-generation counterparts. The main features of the MareNostrum 5 partitions are:

- General Purpose Partition (GPP): This partition has 6480 nodes with Intel Sapphire Rapids processors, each with 112 cores and 256 GB of DDR5 memory. Additionally, there are 72 nodes with Intel Sapphire Rapids 03H-LC processors, each with 112 cores and 128 GB of HBM memory. The nodes are connected by a fat-tree network topology with NDR200 links. The peak performance of this partition is 45.9 PFlops.

- Accelerated Partition (ACC): This partition has 1120 nodes with Intel Sapphire Rapids processors and Nvidia Hopper GPUs, each with 64 cores, 512 GB of DDR5 memory, and 4 GPUs with 64 GB of HBM2 memory. The nodes are connected by a fat-tree network topology with NDR200 links. The peak performance of this partition is 260 PFlops.
- In addition to these, there will be two Next Generation Partitions, namely GPP-NG and ACC-NG. The first one will be based on Nvidia GRACE CPUs. The ACC-GP partition is not fully defined yet so that more information will be provided in the following months.

MareNostrum 5 also provides a high-performance storage system of 248PB net capacity based on SSD/Flash and hard disks, with an aggregated performance of 1.2TB/s on writes and 1.6TB/s on reads. A long-term archive storage solution based on tapes provides an additional 402PB capacity.

7.3 Karolina

IT4Innovations national supercomputing centre in Czech Republic (IT4I@VSB) hosts the petascale system Karolina. Karolina, acquired as part of the EuroHPC Joint Undertaking, was installed in 2021. In the TOP500 list, which evaluates supercomputers in terms of their performance, it ranked 69th worldwide, 19th in Europe, and in the Green500 list of the most energy-efficient supercomputers, it even ranked 8th in 2021. The supercomputer reaches a theoretical peak performance of 15.7 PFlop/s.

The supercomputer consists of 6 main parts:

- a universal part for standard numerical simulations, which consists of approximately 720 computer servers with a theoretical peak performance of 3.8 PFlop/s,
- an accelerated part with 72 servers, and each of them is equipped with 8 GPU accelerators providing a performance of 11.6 PFlop/s for standard HPC simulations and up to 360 PFlop/s for artificial intelligence computations,
- a part designated for large dataset processing that provides a shared memory of as high as 24 TB, and a performance of 74 TFlop/s,
- 36 servers with a performance of 192 TFlop/s are dedicated to providing cloud services,
- a high-speed network to connect all parts as well as individual servers at a speed of up to 200 Gb/s,
- data storage that provides space for 1.4 PB of user data processing and also includes high-speed data storage with a speed of 1 TB/s for simulations as well as computations in the fields of advanced data analysis and artificial intelligence.

IT4I@VSB MAX team has deployed MAX software also as end-user accessible installed modules on Karolina: QuantumESPRESSO, YAMBO and SIESTA are available with current versions and documentation.

(The same packages are also available on the Barbora Czech national system). BigDFT as of now is a work in progress and will be followed by Fleur early 2024.

In addition to providing the MAX codes to the end-users of the Karolina system, IT4I systems are also used to demonstrate the possible energy savings when the system is tuned for a particular running application. This work is performed in collaboration with WP4 which has a dedicated task to energy efficiency evaluation and optimization.

The following figure shows the example of the energy efficiency evaluation of the SIESTA code running on the Karolina machine for one selected use case. Karolina enables the changing the DVFS of the AMD EPYC 7H12 CPUs which are used in the dual-socket servers of the non-accelerated partition. We measured the energy consumption for CPU frequencies from 1.2 - 3.3 GHz, where the latter (3,3 GHz) is the default settings, where CPU is trying to run at turbo frequency as much as cooling of the CPU allows it. We can see that for this particular case we can run CPUs on 2.3 GHz and save 16% of energy without impacting the runtime. If energy is the primal concern one can underclock the CPUs to 1.9 GHz to obtain almost 21% of the energy savings.

Set CPU frequency [GHz]	Real CPU frequency [GHz]	Runtime [s]	Runtime extension	Average power consumption of the CPU [W]	CPUs energy consumption [kJ]	Energy savings for CPUs	Node energy consumption [kJ]	Compute node energy savings	Note
3,30	3,00	436	100%	197,5	172,2	0%	224,5	0%	Default settings
3,20	2,95	434	100%	192,2	167,0	3%	219,1	2%	
3,10	2,89	435	100%	186,3	162,2	6%	214,4	5%	
3,00	2,83	435	100%	181,4	158,0	8%	210,2	6%	
2,90	2,76	435	100%	176,9	154,0	11%	206,3	8%	
2,80	2,68	436	100%	172,5	150,4	13%	202,7	10%	
2,70	2,60	436	100%	168,9	147,4	14%	199,8	11%	
2,60	2,53	437	100%	165,2	144,3	16%	196,7	12%	
2,50	2,45	438	100%	161,8	141,6	18%	194,1	14%	
2,40	2,37	438	100%	158,0	138,4	20%	190,9	15%	
2,30	2,28	438	100%	154,2	135,1	22%	187,6	16%	Optimal settings with no runtime penalty
2,20	2,19	439	101%	150,1	131,9	23%	184,6	18%	
2,10	2,09	441	101%	146,0	128,9	25%	181,8	19%	
2,00	2,00	442	101%	142,7	126,3	27%	179,3	20,1%	
1,90	1,90	444	102%	140,7	124,8	28%	178,1	20,7%	Maximum energy savings
1,80	1,80	447	103%	139,3	124,5	28%	178,2	20,6%	
1,70	1,70	453	104%	138,0	125,0	27%	179,4	20%	
1,60	1,60	458	105%	136,8	125,2	27%	180,1	20%	
1,50	1,50	461	106%	135,7	125,1	27%	180,5	20%	
1,40	1,40	463	106%	134,5	124,6	28%	180,2	20%	
1,30	1,30	469	108%	133,1	124,8	28%	181,0	19%	
1,20	1,20	476	109%	131,7	125,3	27%	182,4	19%	

Fig. 22. Example of the energy efficiency evaluation of the SIESTA code running on the Karolina machine for one selected use case.

This is a very short example of work performed on the EuroHPC Karolina system. More of these evaluations will be presented in the WP4 deliverable D4.2 Report on energy consumption evaluation.

However we feel as Karolina is a production system that enables such experiments at scale, it is important to present these also in the scope of WP3.

7.4 Technical, Authentication and Security Considerations for CI/CD Pipelines

Together with the other CoEs, coordinated by CASTIEL, we are committed to the creation of a common ecosystem where all the codes are integrated in a Continuous Integration/Continuous Deployment pipeline system. As a result of our experiments we reported that the SSH executor is not to be recommended as a good solution. We suggest using Jacamar with technical access and support at sites as the best solution. Jacamar-auth can then be used for authorisation controls, which enables isolation between CI jobs on the runner host. We expect that multiple different authentication and 2FA solutions, together with different supported methods for GitLab runners on sites (possibility to run on login nodes, service accounts for GitLab runners etc) will present a technical barrier to implementation of a unified solution and might require us to provide different solutions for each site. Since EuroHPC by itself currently does not provide such and infrastructure (or strongly suggest adoption of using existing solutions by GEANT, EOSC, EGI and others, such as MyAccessID use at LUMI), we would like to be provided with technical/service access and support to run git runners on target sites within the required project identities.
Alternatively, we would welcome a unified/independent solution that would be provided and would not require direct support from the HE. But solutions where access credentials need to be saved in the GitLab server raise some security considerations: both credentials and runners need to be properly protected. A recommendation to implement a shift-left approach to development security has also been advanced: Gitlab runner should be limited to protected branches, user access to those branches should be limited, a centralised security store to run security controls should implemented and used to manage user credentials to access to EuroHPC clusters based on who executes the runner etc. We are looking forward to comparing our results with other CoEs and representatives of other HE in order to ensure an effective availability of the CI/CD for all the MAX codes in the medium-short term.

8 Conclusions

Within WP3, this first phase of the project focused mainly on the definition of a common procedure for the benchmarking and profiling activity, the choice and install of tools for this purpose, the preparation of xml inputs and on the porting of MAX codes to Leonardo. Then, we started the benchmarking and profiling activity on CINECA's newest cluster, with the aim to characterise their potential and bottlenecks towards exascaling. In collaboration with CASTIEL and the other CoEs we also started working to

integrate a unified approach of CI/CD that would permit monitoring the status and progress of the lighthouse applications.

In order to start working on code characterization as soon as the machine and the codes were ready, we adopted a top-down approach for the implementation of the JUBE ecosystem of scripts. We started benchmarking MAX codes with less transversal xml files, tailored for each application. Then, we are progressively outsourcing tags and variables to achieve the desired level of generality depicted in the first part of this deliverable. By adopting this top-down approach, we could start our activity with less general scripts, however suitable to ensure reproducibility of results and a clear structure for the storage of outputs. As it is functional to the collaboration among developers and HPC experts, the improvement of flexibility and generality of our JUBE ecosystem will be one important effort of the next months.

The early analysis done so far provides already interesting inputs for the optimization of MAX codes and for further investigation of the bottlenecks arised. We will pursue code characterization and measurement, in synergy with the objectives of other WPs. The access to more EuroHPC machines, beyond Leonardo and LUMI, would offer the possibility to understand performance on different hardwares and networks. Thanks to the deployment of MAX codes on other clusters, we could address more comprehensively hot topics for scaling on heterogeneous architectures, such as communications with GPU awareness, the integration of codes with workflow managers to enable dynamic scheduling and improve efficiency against workload imbalances at the largest scale. We expect that these investigations will be fostered by the automatization of data production, starting from the benchmark to the visualisation of results.

By taking advantage of the data format standardisation obtained with JUBE, we intend to organise the benchmark repository so that the data can be easily extracted for visualisation; in our view, this operation would be automatized as well with appropriate tools. The effort to unify and automate data visualisation will streamline the dissemination of results and their inspection by production users, thus contributing to minimising the waste of computational resources.