D4.5

Final report on code profiling and bottleneck identification

F. Affinito, U. Alekseeva, A. Degomme, P. Delugas, A. Garcia, A. Kozhevnikov, and N. Spallanzani

Due date of deliverable: 30/11/2021
Actual submission date: 30/11/2021

Lead beneficiary: CINECA (participant number 8)
Dissemination level: PU - Public

Document information

Project acronym:	MaX
Project full title:	Materials Design at the Exascale
Research Action Project type:	European Centre of Excellence in materials modelling, simulations and design
EC Grant agreement no.:	824143
Project starting / end date:	01/12/2018 (month 1) / 31/05/2022 (month 42)
Website:	www.max-centre.eu
Deliverable No.:	D4.5

Authors: F. Affinito, U. Alekseeva, A. Degomme, P. Delugas, A. Garcia, A. Kozhevnikov, and N. Spallanzani

To be cited as: F. Affinito et al., (2021): Final report on code profiling and bottleneck identification. Deliverable D4.5 of the H2020 project MaX (final version as of 30/11/2021). EC grant agreement no: 824143, CINECA, Casalecchio di Reno (BO), Italy.

Disclaimer:

D4.5 Final report on code profiling and bottleneck identification

Content

Executive Summary

This final report presents a general summary of the profiling and benchmarking activities in WP4. Thanks to this task we have at first identified bottlenecks and inefficiencies in the flagship codes; and we continue to provide the developers a check on their performance progress.

Our screening is based on a defined set of scientific test cases on which we benchmark the codes. This collection contains the main types of calculation that can be executed by MaX codes in supercomputing centres. The tested systems have sizes that span several orders of magnitude. We can thus assess the efficiency of codes in exploiting as much computational power as possible, either in the case of very large systems (many thousands of atoms); or performing High Throughput Computations (HTC) on small and medium-size systems, to quickly explore wide configuration spaces.

The bottlenecks identified by the benchmarks were in many cases unexpected or overlooked. They were revealed by our tests thanks to their very large size, or because -with the intent to mimic the resources accessible in exascale machines- we executed them with an (at the moment) unusually large amount of computing resources.

Although specific to each code, it has been possible to broadly categorize most of the bottlenecks in a few case types:

- Particularly in the first part of the activity, many bottlenecks were represented by code parts whose missing or inefficient parallelization had been made apparent only by the usage of a very large number of nodes. The work on these residual parallelization parts has been done both using the still existing MPI parallelization, improving the hybrid MPI + OpenMP parallelism and for the GPU ready version offloading part of the calculation to the accelerators.
- The porting to accelerators has reduced significantly the number of MPI tasks used for the calculations and code parts that relied on the distribution over a large number of MPI ranks have been refactored to exploit GPUs when available.
- In other cases, bottlenecks are more structural and inherent to the algorithm or the methodology adopted for the calculation. This has motivated the exploration of alternative algorithms or the adoption of domain-specific libraries targeting the issues more efficiently. In general, the optimization of the codes for the heterogeneous architectures has involved a detailed profiling activity on all code and of many of the case studies.
- The optimization of the I/O has been a general issue that has involved many of the codes and benchmarks.

Later runs of the benchmarks helped us monitor and assess the work done to eliminate or mitigate the bottlenecks. In this final report, we present for each code the current status, the ongoing work, and the perspectives.

- **BigDFT**
  - combined usage of containers, AiiDA, and PyBigDFT for the fast deployment and analysis of benchmarks;

  - report of the latest tests on the Fugaku supercomputer with a comparison of the code general performance on this architecture with the tests previously collected.
- **CP2K**
  - general evolution of the standard test for the linear scaling algorithm using 256 water molecules; performance improvement in the RPA test with 128 water molecules using the COSMA and COSTA libraries to perform the parallel linear algebra on GPUs.
- **FLEUR**
  - General screening of the benchmark time to solution in different new architectures;
  - The progress done in scalability for the matrix and charge setup parts of the code, where a significant performance improvement has been achieved refactoring the access to the muffin tin coefficients;
  - The performance improvement achieved for the non-collinear magnetism case after the optimization of the data communication for this case.
- **Quantum ESPRESSO** reports:
  - An update on the GrCoIr benchmark with the execution times obtained using the A100 GPUs;
  - The better scalability of the RMM-DIIS iterative eigensolver. This algorithm avoids the dense diagonalization bottleneck of the Davidson algorithm;
  - The performance of the completed porting of cp.x for GPUs.
- For **Siesta** we report the status of the main bottlenecks for the code.
  - For what concerns the performance of distributed linear algebra, important improvements have been achieved with the usage of the ELPA specific library. The availability of more general solvers in future versions of the library will allow for the possibility to avoid the Cholesky decomposition bottleneck.
  - For what concerns the linear solvers some algorithmic improvements have been implemented in the framework of WP3 and are starting to be tested.
- For **YAMBO** we report:
  - Performance improvements in the dipole part as a result of the streamlined data communication between nodes and host-device memory transfers.
  - The results of the work on the I/O optimization.

The performance screening will continue in the next months. New developments of WP1, WP2, and WP3 will be included in the oncoming code versions; and it will be also necessary to monitor the possible novelties for what concerns alternative architectures that should become accessible in the next future.

For what concerns the profiling activities, as these have become more inherent to the co-design part of activities, we have decided to present them in the corresponding report.

Introduction

One of the main targets of the work done during MaX phase-2 is to step up and make more portable the performance of the flagship codes so as to make them able to exploit the enormous computational power that will be available in exascale supercomputers.

In view of this target, WP4 has planned and coordinated the analysis and monitoring of the performance of the MaX codes. This task has involved code developers, materials science specialists, and supercomputing experts. Their actions have been coordinated with those of the codesign task in WP4, and of the code development in WPs 1, 2, and 3. The task has also interacted at many stages with WPs 5 and 6.

The first activity in the task was the identification of the set of scientific test cases to use as benchmarks of the flagship codes. The cases were chosen so as to include all the main calculation types, and cover several orders of magnitude of system sizes and computational loads.

The benchmarks are publicly available in a repository on GitLab[1] together with all results of the tests executed during this activity.

At the first stages of the project, the inspection of the benchmark results was very helpful for identifying many significant bottlenecks. Some of these criticalities had been unexpected or overlooked, and they became more evident either because of the very large size of some of the benchmarks or, in other cases, only because of the unusually large amount of computational resources used to execute them.

The identified bottlenecks and issues were obviously specific to each code, but some general patterns could be identified and rationalised:

- non-parallelized code parts that became inefficient when the number of MPI ranks was pushed high;
- data structures unfit for applications distributed on many MPI ranks;
- intrinsically weakly scaling algorithms (e.g., dense diagonalization of distributed large matrices);
- I/O bottlenecks caused by the very large size of the benchmarks or by the usage of a very large number on MPI ranks.

On the basis of the outcomes of the benchmarks, we have planned some coding actions for the elimination or mitigation of the identified bottlenecks.

The first results of such actions were already visible in the D4.3[2] report where we reported the benchmarks on the second release of the MaX codes. In this second deliverable, many of the benchmarks were also executed on heterogeneous architectures based on GPUs.

This allowed us to compare the performance and the computation costs of the homogeneous multicore architectures with those of the accelerated heterogeneous systems. At the moment the GPU accelerated systems perform systematically faster than the traditional multicore homogeneous

[1] https://gitlab.com/max-centre/benchmarks
[2] http://www.max-centre.eu/sites/default/files/D4.4_First%20report%20on%20co-design%20actions.pdf

systems. A comparison in terms of computation costs is instead more complex. Using a simplified approach, considering, e.g., the cost in node-hours, the cost comparison would be again in favour of the heterogeneous systems. A more thorough analysis taking into account the effective energy consumption of calculations has been done as part of the codesign task of WP4. The results are reported in deliverable D4.6. These measurements are more complex and are hardly feasible for the large size calculations involved in most of the benchmarks.

This third deliverable presents the status of the benchmarks at month 36 of the project. Clearly, while many of the issues have been solved, others have been mitigated by adopting alternative solutions or algorithms. Beyond the solutions identified with the first benchmarking campaign, the developers have continued to work on new solutions that could further improve on the performance of the codes. For example, some of the new algorithms developed in WP3 have been applied for the execution of the benchmarks, and have in many cases brought significant performance improvements.

The results of the benchmarks not only reflect the evolution of the codes; they also follow the improvements of the whole ecosystem: hardware, software stack, and domain-specific libraries. This is a clear indication of how good the readiness of the codes is to follow the evolution of the supercomputing technology.

In the next Sections we will discuss the results of the benchmarking activities for each of the MaX codes, and finally conclude by discussing perspectives and lessons learnt during the profiling and benchmarking campaigns.

BigDFT

Most of the work over the past months in BigDFT has been directed at bringing new features and capacities, notably around the *fragmen*t approach, to the PyBigDFT library, or to the Sirius integration. The performance of the code itself was not the main focus of this development, even if every new cluster and architecture offer new challenges and insights. For instance, the Fugaku supercomputer, with ARM processors and SVE support, was a target of choice in our developments. In this regard, the BOAST auto-tuning framework has been extended to generate SVE instructions when generating vector instructions (see D4.6).

New container architecture (modularity, portability, performance)

BigDFT has been distributed in Docker containers for several years. These are available for download on dockerhub[3] or on the Nvidia NGC initiative[4]. This kind of distribution provides a highly optimized version of the code (with support for GPUs, MKL, and several MPI libraries). Nvidia HPC container maker scripts are used to build the Docker or Singularity recipes. These can be further tuned to accommodate several options. As the containers allow for fast and controlled deployment of the executable on most of the existing platforms, they provide one ideal tool to test the performance of the application on various setups. The use of containers for the benchmarking has also been

[3] https://hub.docker.com/
[4] https://catalog.ngc.nvidia.com/containers

streamlined by using AiiDA workflows and PyBigDFT tools to handle the submission and the analysis part of the benchmarks.

This part of the container generation system has been completely rewritten recently, to provide better flexibility and portability. Still using HPC container maker to dynamically generate the docker or singularity recipes using python scripts, it now supports Intel OneAPI containers for x86 platforms, as well as Nvidia CUDA containers. The Nvidia containers provide more portability, as they are available for ARM and IBM Power platforms, but the OneAPI container comes with Intel compilers, MKL libraries and Intel MPI, which can provide better performances on these architectures.

The main goal of the redesign is to allow for easy generation of several SDK flavours for developers willing to quickly validate BigDFT with multiple systems, compilers, MPI libraries, and linear algebra libraries. Users can also quickly test several versions of BigDFT on a new cluster to test which provides the best performance.

The combination of easy container deployment and AiiDA-based benchmarking workflows makes it easy to characterise new systems and analyse possible bottlenecks.

Benchmarks and bottlenecks

BigDFT has been successfully deployed on the Fugaku supercomputer. This was not done with container technology as the Singularity installation on Fugaku was not functional at this time, and the experiments were made using the proprietary Fujitsu compilers, not available in container format. During the initial benchmarking process, an important performance bottleneck was found in analysing the profiling output and comparing it with the same set of benchmarks executed on other supercomputers. The issue was tracked down to the use of internal profiling routines themselves, which were not previously known to abnormally stress the processors in other systems. These routines can be easily deactivated, hence mitigating the importance of this issue, but causing losses in further retrieval of profiling information. This shows the importance of comparing benchmarks on various platforms and keeping a centralised repository with results on different systems, in order to quickly find such issues and mitigate them.

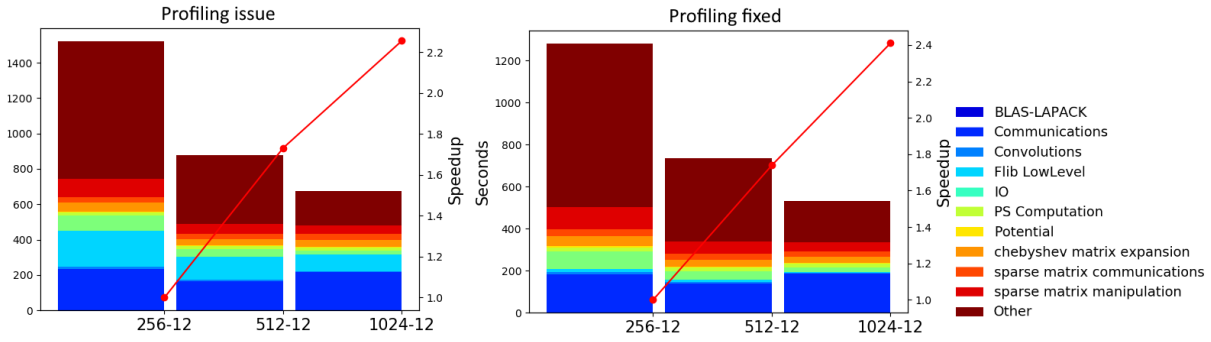

Fig. 1: Fugaku performance with (left) and without (right) advanced profiling routines for monomer input file, with 64-128-256 nodes (4 MPI processes per node, 12 OpenMP threads per MPI process). Tracing is still activated, allowing to get most of the information and easily plot these using Futile python tools, even if detailed memory analysis and tracing are not available anymore. Improvement over previous runs is 16-22%, with scaling improved. Flib lowlevel (light blue) overhead is now negligible.

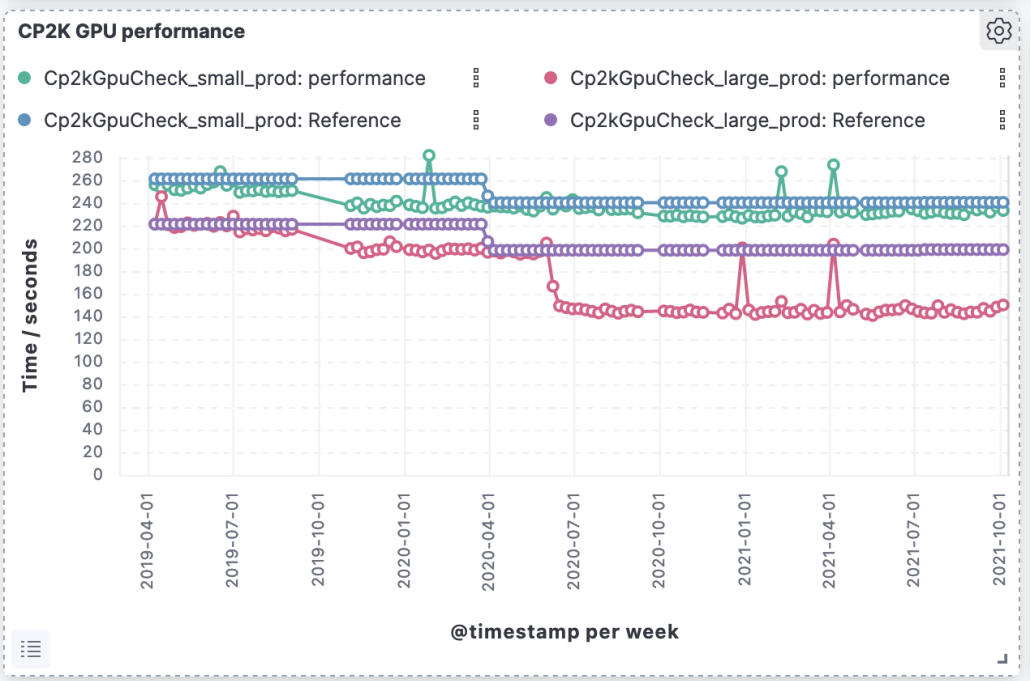

Fig. 2: Performance regression test of CP2K executed daily at CSCS since March 2019. The test computes the ground state energy of 256 water molecules in the linear scaling mode. The test is executed on 6 (small) and 16 (large) GPU nodes of Piz Daint. During the past years, the scaling of the CP2K code and its libraries have been considerably improved (~35% reduction in time-to-solution of the large run).

CP2K

CP2K is a large community code containing around ~1M lines of source code to support various types of calculations. In the context of the project, we focused on the performance optimization of the linear-scaling and RPA types of calculations. The work on linear scaling is covered in WP2 where we explored the just-in-time compilation (JIT) of the GPU kernels for the DBCSR library. The evolution of the code performance for the linear-scaling regime is shown on Fig. 2, where we report the daily measurements of CP2K execution on Piz Daint since March 2019.

To enable the CP2K scientific demonstrator that proves the readiness of the code for the (pre-)exascale systems, we have worked on the optimization of large tall-and-skinny matrix-matrix multiplications. This operation was identified as a major bottleneck at the beginning of the project. The standard tool for such distributed matrix multiplication is de-facto a ScaLAPACK library and its HPC implementation by Intel MKL that works only on CPUs and Intel accelerators. To improve this situation we have been working on the open-source communication-optimal multiplication library COSMA and its ScaLAPACK interface (including COSTA library for the fast data reshuffling). The latter allows for the use of COSMA in CP2K without introducing any change in the CP2K code. Fixing all the corner cases and passing all of the CP2K's regression tests has been a challenging task which is now complete – CP2K-8.1 and higher supports COSMA linking. The final benchmark of the RPA calculations of 128 water molecules with CP2K code on [128-1024] nodes of Piz Daint are demonstrated in Fig.3. As a final remark, it has to be noted that the COSMA library has both NVIDIA and AMD GPU backends and it is going to provide the performance of CP2K code on the LUMI platform from day zero.

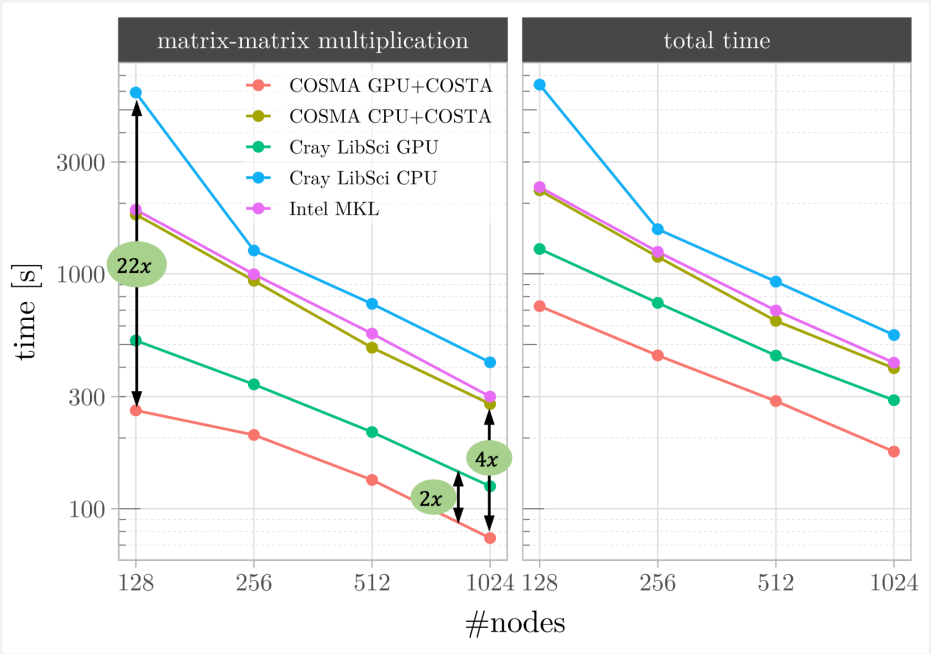

Fig. 3: Performance of the CP2K code in the RPA calculation of 128 water molecules. The benchmark was run on the Piz Daint GPU partition, using different matrix multiplication backends. COSMA+COSTA achieves the best performance on both CPUs and GPUs. Left panel: time spent in the matrix-matrix multiplication part (construction of the free Green's function). Right panel: total execution time.

FLEUR

This Section reports about the benchmarking activities on the FLEUR code and its performance improvements. As elaborated in the executive summary of Deliverable D4.2[5], our codes are very versatile, and allow for the calculation of many properties in diverse setups. These richness and flexibility imply that a given code possesses many code paths, which may be chosen depending on the nature of the simulation. The large benchmarks presented in Deliverable D4.2 reflect the most general case (bulk non-magnetic unit cells with and without inversion symmetry with local orbitals without spin-orbit coupling) which embraces the routines that are present in almost any calculation with FLEUR. This is the most important direction of the work and we will report the progress in this kind of calculation. Nevertheless, during the execution of the project, mainly under the influence of work package WP6 (where the simulations of complex particle-like magnetic structures are being performed) another very significant case - non-collinear magnetic systems with spin-orbit coupling - also came into the performance investigation. Benchmarks on other architectures such as Arm, AMD, and Nvidia GPUs are shown at the end.

5 http://www.max-centre.eu/sites/default/files/D4.2%20First%20report%20on%20code%20profiling%20and%20bottleneck%20identification%2C%20structured%20plan%20of%20forward%20activities.pdf

General case: matrix setup & new charge

The muffin-tin matching coefficients can be represented as matrices, one matrix for every atom. One dimension of such a matrix is the number of basis functions (which is usually about 100 times the number of atoms) and the other is the number of angular momentum indices (which is usually about 100). These coefficients are needed in the calculation of the non-spherical part of the Hamiltonian matrix, charge densities, and contribution from the local orbitals (LO). It is therefore very important for the performance of the code that reading and writing to the arrays representing the muffin-tin coefficients is done according to the way they are stored. After tidying up all the accesses to these matrices, we have seen the expected improvement in the non-spherical setup, LO-setup, and new charge generation (Tab. 1). Additionally, for the case of real matrices, OpenMP pragmas were added in the non-spherical matrix setup. The performance evolution of the FLEUR code during MaX phase-2 can be seen in Figs. 4 and 5.

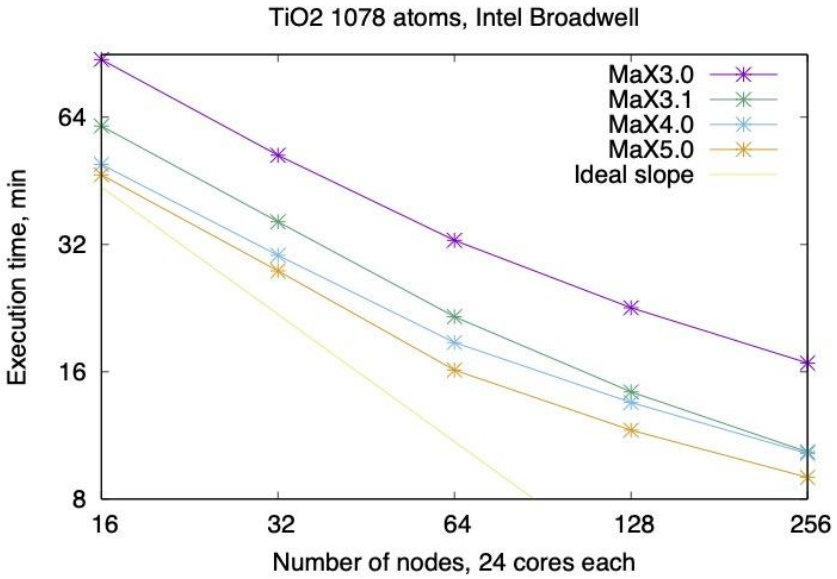

Fig. 4: Performance evolution of the benchmark TiO2 1000 atoms (complex matrix). The improvement from the Release MaX3.0 to the MaX3.1 is mostly due to the better data layout, from the Release MaX3.1 to the MaX 4.0 is mainly due to the optimization of the spherical setup, from the Release MaX4.0 to the MaX5.0 is mainly due to the memory access optimization to the muffin-tin coefficients.

	non-spherical setup, sec		local orbitals, sec		new charge generation, sec	
# nodes	before	after	before	after	before	after
16	372.63	304.74	43.03	40.16	116.15	62.70
32	257.53	194.26	42.75	41.46	99.51	42.65
64	196.38	138.39	42.61	42.42	98.16	39.79
128	169.29	108.39	42.08	40.05	89.7	32.74

Table 1: Runtime improvement of the non-spherical matrix setup, calculation of local orbitals and new charge generation due to the access optimization to the muffin-tin coefficients. The measurements are performed on 16, 32, 64 and 128 Intel Skylake nodes (CLAIX, RWTH Aachen University), 48 cores/each. Benchmark: TiO2 1078 atoms.

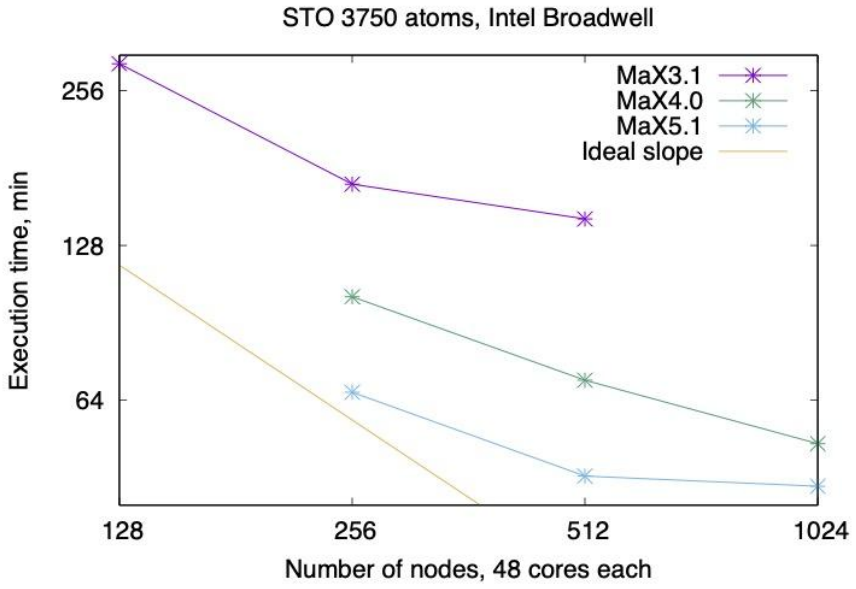

Fig. 5: Performance evolution of the benchmark STO 3750 atoms (real matrix). This calculation was not possible with the version MaX3.0. The measurements with the version MaX3.1 were done on the CLAIX machine during the installation phase, so it was very unstable. The improvement from the Release MaX4.0 to the MaX5.1 is due to the memory access optimization to the muffin-tin coefficients and added multi-threading in the non-spherical matrix setup; both versions were tested on the SuperMUC-NG machine.

Non-collinear magnetism case

Considering a system with a given number of atoms, the non-collinear magnetic simulation requires four times more memory and eight times more time. These differences come from the increased Hamiltonian and overlap matrices: these dense complex Hermitian matrices double in size in both directions. Additionally, there are some new routines for the matrix setup which have not been considered from the performance point of view. Before attempting the fine-tuning of performance for large systems, missing parallelism was added in the subroutine hsmt_distspins. The parallelization of this specific routine, never considered before, made the muffin-tin setup more than 5 times faster and more than halved the whole execution time (Tab.2). After that, the tests with the bigger systems (256, 512 and 1024 atoms) became possible. These improvements are included in the code from the MaX 5.0 Release.

	hsmt_distspins, sec	Total MT, sec	Total iteration, sec
before	249.49	363.65	545.28
after	11.34	65.92	230.89
speedup	22.0	5.5	2.36

Table 2: Runtime improvement after the parallelisation of the subroutine “hsmt_distspins” is shown together with a comparison of the whole muffin-tin setup (MT) and the whole self-consistency interaction (last column). Measurements were done with a small magnetic test case MnGe with 128 atoms on one node of CLAIX, Aachen (Intel Skylake, 48 cores).

The data layouts for the setup of the matrices and for the diagonalization are different: 1-column cyclic for the first and block-cyclic for the second, and the matrices are redistributed between the setup and the diagonalization via a ScaLAPACK call. This redistribution did not take much time in the non-magnetic and collinear case, while it turned out to be a serious bottleneck in the non-collinear case. Here, for the spin-up-spin-down part of the Hamiltonian matrix, a transposed upper half of the distributed matrix needs to be added to the lower part, which requires some reshuffle of the data since the data are created column-wise but are expected row-wise. This was done via a call to the pzgeadd subroutine. With the 1-column cyclic data layout, the matrix elements were sent separately which drastically reduced the performance of the code for the large systems. After implementing a custom routine that first gathers data to be sent and then transfers them as a package, the times went significantly down (Tab.3). These and some other minor optimizations made it possible to include large non-collinear magnetic systems with over 512, 1024 and 1600 atoms (Fig.6) in the benchmark set and made possible calculations of Bloch points and other complex magnetic structures as it will be shown in WP6.

	128 nodes	256 nodes	512 nodes
before	125.79 sec	218.81 sec	647.32 sec
after	14.97 sec	39.01 sec	32.84 sec
speedup	8.40	5.61	19.71

Table 3: Runtime improvement of the matrix redistribution after implementing the custom communication routine. Test case: MnGe supercell 4x4x4 with 1024 atoms. Hardware: Intel Skylake, 48 cores/node (SuperMUC-NG, Munich).

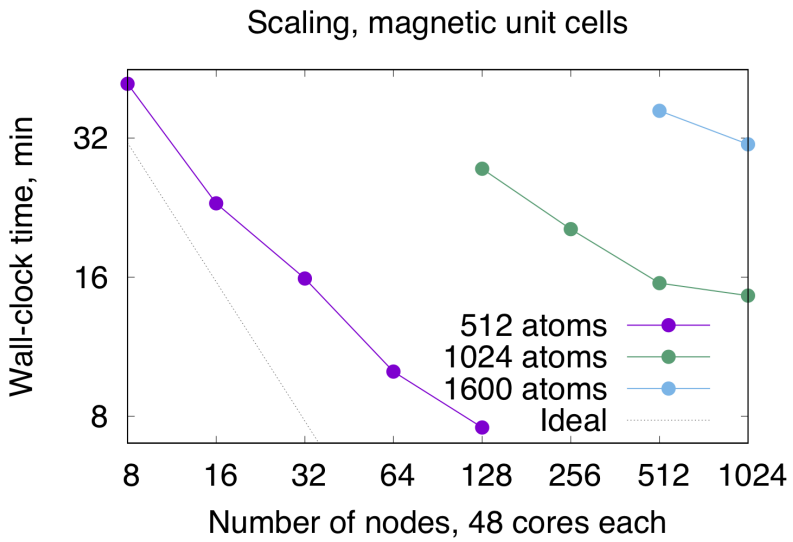

Fig. 6: Scaling of three large non-collinear magnetic benchmarks with 512 (magenta), 1024 (green) and 1024 atoms (blue). Calculations were performed with the MaX Release 5.0 on the SuperMUC-NG supercomputer.

Benchmarks on other architectures

We also benchmarked the code on other architectures such as AMD, Arm, and Nvidia GPUs and compared the performance of the code with that on the Intel architecture. Please note that in case of AMD and Arm CPUs there was no special optimization tuning for these architectures (except appropriate compiler flags). For the AMD, the performance comparison of different combinations of hardware and software, i.e. one case with Intel Broadwell CPU + Intel compiler + IntelMPI + Intel MKL, another one with AMD Epyc CPU + gcc + BLIS + libFLAME + ScaLAPACK + ParaStationMPI, and a mixed one with AMD Epyc CPU + Intel compiler + IntelMPI + Intel MKL, show similar performance on machines with similar peak performance (Fig.7).

On the JUAWEI cluster (Arm 32 cores Cortex A-72 architecture) we were not able to use the ScaLAPACK library: the code compiled but the execution ran into the time limit event after 20 times increase of the requested time. This is the reason why we have tested only the highest level of the MPI parallelization, i.e. over k-points (Tab.4). We also compared the execution on one MPI rank with the threads spread over the whole node for two cases, once for the Arm Cortex A-72 node and then for the Intel Broadwell. These nodes have comparable peak performance: 614 and 840 GFlops for the Arm and Intel machines, respectively (Tab.5). It is nice to see that the execution times of the whole iteration are also similar, though the distribution of the time between different code parts varies considerably.

The work on the porting of the matrix setup to GPUs has been continued. The previous implementation with CUDA Fortran was replaced with OpenACC. Two benchmarks, CuAg with 256 atoms and GaAs with 512 atoms were tested on the JURECA-DC machine in Jülich, Germany. Simulations were performed on 1, 2, and 4 nodes, each containing either 2 AMD Epyc CPUs or 4 NVIDIA A100 GPUs (Tab.6).

Test case	# MPI	potential	mat. setup	diagonal.	new charge	Whole iter.
NaCl 64 atoms 2 k-points	1	20.38	35.12	316.63	14.78	392.28
	2	36.56	59.86	176.07	17.93	299.6
AuAg 108 atoms 4 k-points	1	56.83	521.15	5085.2	199.14	5887.83
	2	71.49	902.93	5057.04	232.45	6323.41
	4	95.65	1799.93	3441.58	366.6	5813.61

Table 4: Benchmarks on Arm, GCC + OpenBLAS+OpenMPI, MaX Release 5.0. Different parts of the code (potential generation, setup of the matrices, diagonalization and generation of the new charge) is shown together with the whole self-consistency iteration.

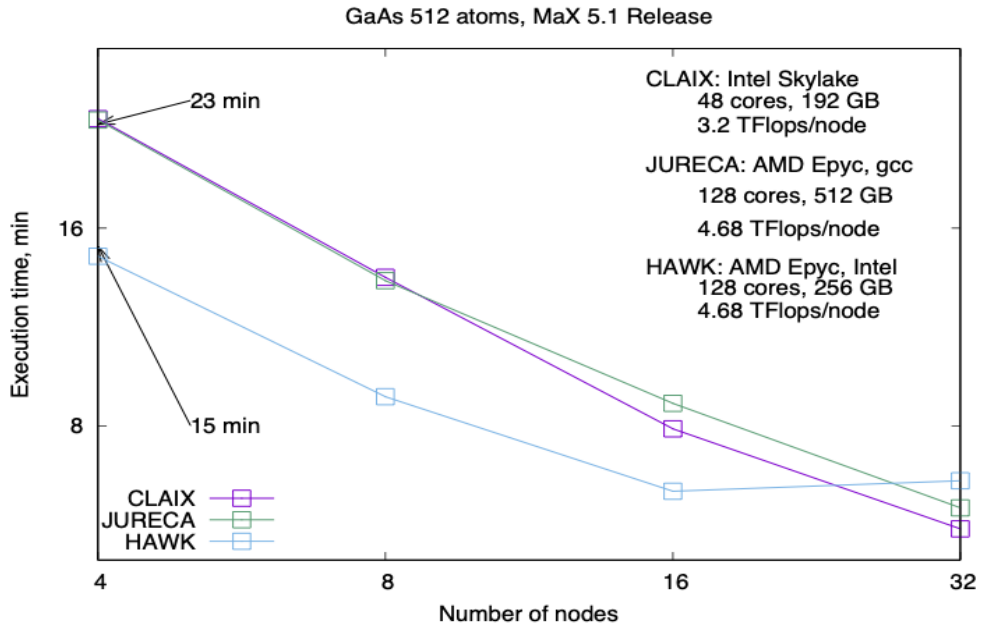

Fig. 7: Comparison of the performance of the FLEUR code for the same benchmark with different combinations of hardware and software: i) Intel Chip + Intel compiler + IntelMPI + Intel MKL (CLAIX, magenta), ii) AMD Chip + AMD SW (JURECA, green), iii) AMD Chip + Intel SW (HAWK, blue). Test case GaAs 512 atoms. Code Release MaX 5.1. More information about the machines is on the plot.

CPU	# threads	potential	mat. setup	diagonal.	new charge	Whole iter.
Intel	24	2.95	3.28	166.43	3.88	178.12
Arm	64	20.4	17.16	159.42	9.99	211.6

Table 5: Benchmarks on Arm Cortex A-72 node with 64 cores vs. Intel Broadwell with 24 cores, with peak performances of 614 and 840 GFlops respectively. Test case: NaCl 64 atoms. Different parts of the code (potential generation, setup of the matrices, diagonalization and generation of the new charge) is shown together with the whole self-consistency iteration.

Test case	CuAg, 256 atoms			GaAs, 512 atoms	
# nodes	1	2	4	2	4
CPU	200.92	109.62	61.92	1465.56	750.53
GPU	22.05	15.04	11.21	244.43	189.95

Table 6: Runtime comparison between execution on CPU (AMD Epyc, 2 x 64 cores/ node) and GPU (4 x NVIDIA A100/node) for the matrix setup. Two benchmarks are presented: CuAg 256 atoms and GaAs 512 atoms. Simulations were done on 1,2,4 and 2,4 nodes respectively. Only measurements of the matrix setup are shown. FLEUR version MaX 5.1.

Quantum ESPRESSO

In the last year, the development work in Quantum ESPRESSO concerning performance was mainly aimed at the completion of the GPU acceleration of cp.x, offloading some residual parts of the code that had become significant bottlenecks. The residual acceleration has involved the Gram-Schmidt orthogonalization, the force and stress computation, and the offloading to GPUs of the computation of the density functional terms. This has allowed for a significant improvement of the performance, with also a large reduction in the computation cost in terms of node-hours.

For what concerns pw.x, the main novelties come from the work on alternative algorithms for the iterative diagonalization of the Hamiltonian operator. All of these algorithms aim at removing or reducing the usage of dense diagonalization of large matrices that is the main bottleneck of the Davidson algorithm for large systems. The dense diagonalization bottleneck is also significantly alleviated by the increase of device memory provided by new GPU cards, because this allows for the allocation and efficient diagonalization of larger matrices.

In the following parts of this Section we present the benchmark data illustrating the achievements listed above.

Update on pw.x: new hardware and algorithms

Graphene-Co-Ir benchmark on A100 GPU cards

The increase of device memory in A100 NVidia cards (40 or 80 Gb) typically leads to a significant improvement of the pw.x performance on CUDA systems. In particular, we find a significant mitigation of the memory limitations that have hindered the scaling of very large tests.

The 80 GB memory card allows for the execution of the Graphene-Co-Ir benchmark with a minimum number of 4 A100 GPUs.

Using from 1 up to 12 nodes, each equipped with 8 A100 GPU cards and 64 AMD EPYC cores, we have been able to distribute the calculation up to the maximum possible number of pools; using for each

pool 4 or 8 GPUs. Thanks to the full pools distribution we obtain (Fig. 8) an almost ideal parallel efficiency up to 6 nodes (12 pools, 4GPUs per pool). Over 6 nodes further speedup is obtained using intrapool parallelism. For this, passing from 4 to 8 GPUs per pool, we obtain an average speedup of about 1.5 (Fig. 9).

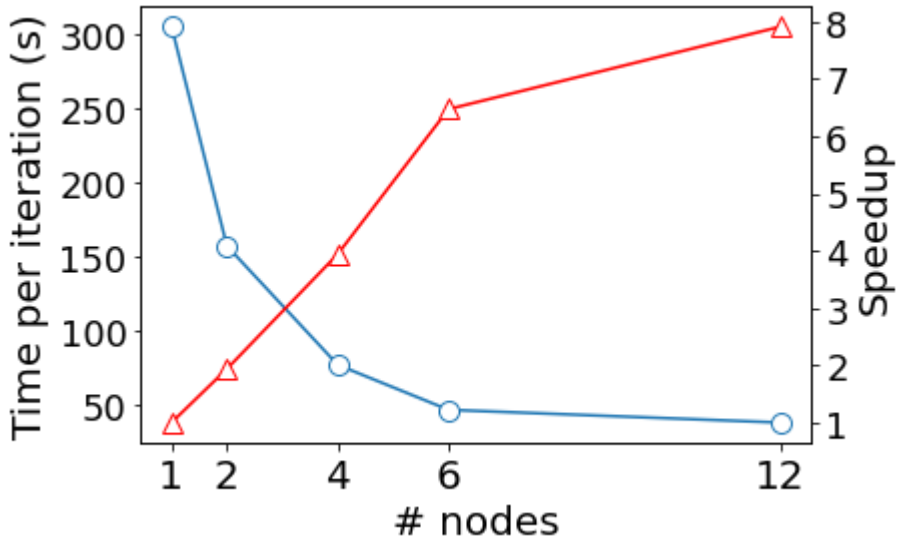

Fig. 8: Average time per iteration and speedup for the GrCoIr benchmark run on an A100 cluster. Each node is provided with 8 A100 NVidia Cards.

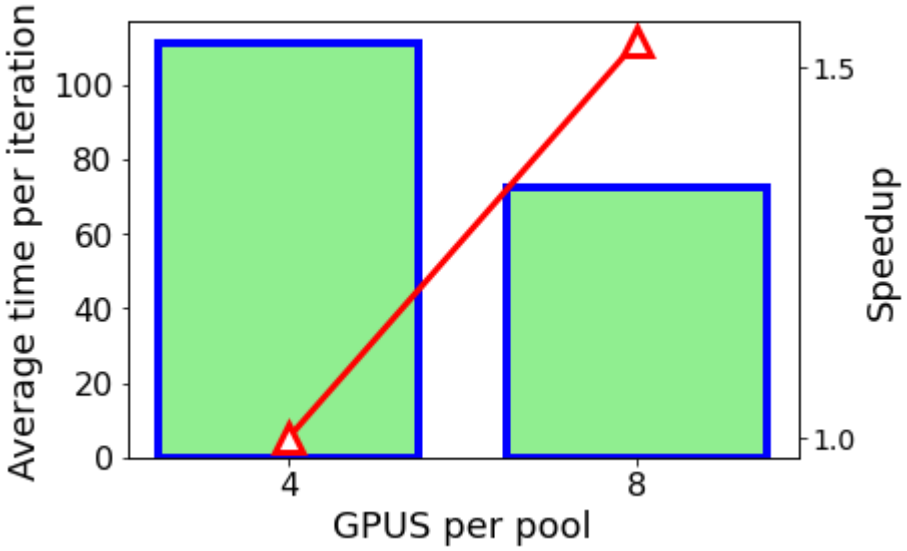

Fig. 9: Speedup for parallelism inside each pool. The times per iteration and speedup have been obtained considering an average of the different executions plotted in Fig. 8.

CNTPOR8 benchmark: scaling improvement with the new RMM-DIIS algorithm

The main algorithm used in pw.x for the iterative diagonalization of the Hamiltonian is the Davidson algorithm. This approach performs at each step the diagonalization of a dense matrix whose order is a multiple of the number of bands (proportional to the number of simulated electrons). The algorithm is very efficient as long as the number of bands is small and starts performing poorly when the number of bands is on the order of thousands. For large calculations in massively parallel machines, the dense diagonalization becomes the bottleneck because of the intrinsic limitation posed by the performance of state-of-the-art libraries for parallel diagonalization (e.g. SCALAPACK, ELPA, etc.).

To reduce the impact on performance we have thus started to explore alternative approaches that avoid or make less frequent use of dense parallelization of large matrices. At the moment, the KS_Solver library of QE features 4 alternative iterative algorithms CG, PPCG, ParO and RMM-DIIS.

While reducing the usage of dense linear algebra, the first three approaches (CG, PPCG, and ParO) still need to be integrated with the ongoing work on band-group parallelism in order to become effectively competitive with Davidson in very large benchmarks. They are already very useful in cases where the Davidson solver fails, or to stabilize the RMM-DIIS solver. This latter instead already provides a better scaling alternative to the Davidson algorithm. More work in the WP3 context is ongoing to solve the stability issues that, in the current version, are avoided by using it in combination with other solvers.

We report here (Fig. 10) our results for RMM-DIIS in the CNTPOR8 benchmark. The calculations were executed on the A3 partition of the Marconi cluster of CINECA comparing the speedup obtained with the new iterative solver against the Davidson performance. Interestingly, the RMM-DIIS algorithm is generally as efficient as the Davidson and scales better.

Acceleration of cp.x using OpenACC and CUDA-Fortran

The porting of the cp.x quantum engine has been completed and will be released for production in version 7.0 of Q.E. For what concerns mathematical libraries and low-level routines the porting reuses the already accelerated CUDA-Fortran code parts. At the higher level, similar to what is happening for the high-level drivers in pw.x, the GPU porting has been done using the OpenACC model. This avoids the duplication of global variable references and the splitting of the high-level routines. A detailed description of this porting is given in other deliverables. Here we report the first set of benchmark tests for the accelerated version.

The benchmarks and the profiling demonstrate that the porting has already reached an acceptable level of performance. Ongoing work for optimizing device-host data will likely improve the performance.

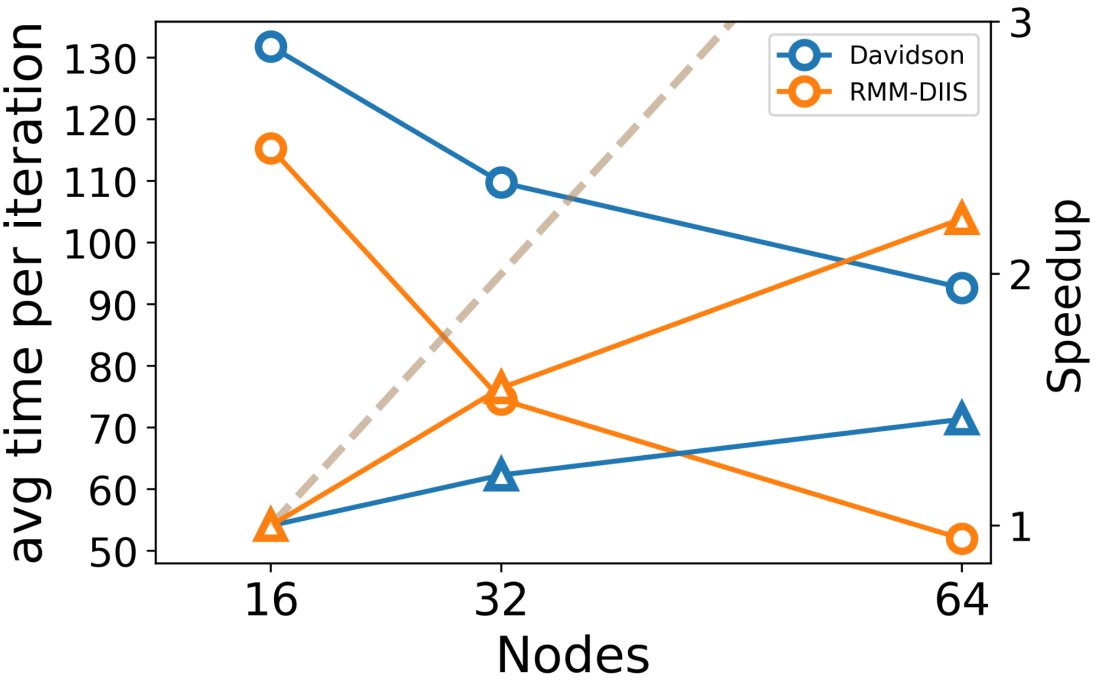

Fig. 10: Comparative performance of Davidson and RMM-DIIS algorithms for the CNTPOR8 benchmark run on MarconiA3@CINECA. Each node is provided with 48 Intel SkyLake cores.

W256 benchmark

As a general baseline benchmark for the accelerated cp.x performance we report a standard calculation that includes both the initialization and 50 steps of molecular dynamics (MD) for a system made of 256 water molecules. The benchmark has been run on the Marconi100 cluster of CINECA. Each node of the cluster has 4 NVIDIA Volta V100 GPUs and 32 IBM Power9 AC922 cores.

The speedup of the time to solution is reported in Fig. 11, while Fig. 12 shows the extrapolated cost in node-hours for 1000 MD steps.

The benchmark shows how the performance at a low number of nodes is already satisfactory. Improvements may be expected by the ongoing work on the optimization of the device-host memory synchronization.

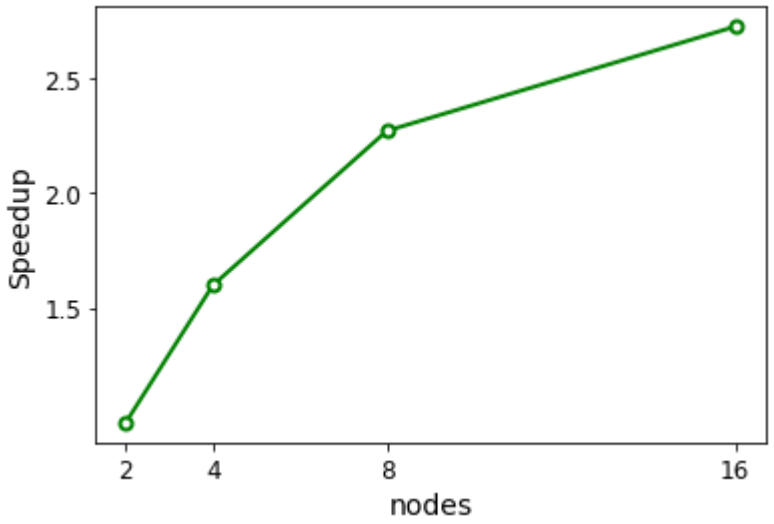

Fig. 11: Speedup for the W256 test with the accelerated versions of cp.x.

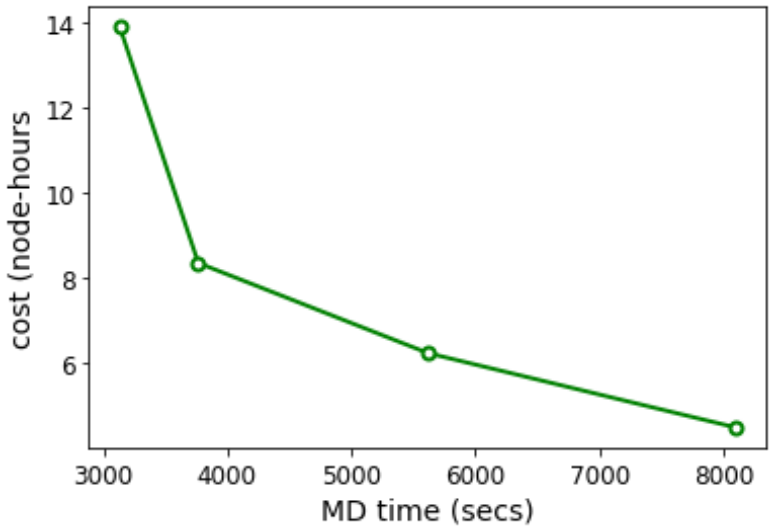

Fig. 12: Computation cost in node-hours for a 1000 steps MD simulation of the W256 system.

ZrO_2 benchmark

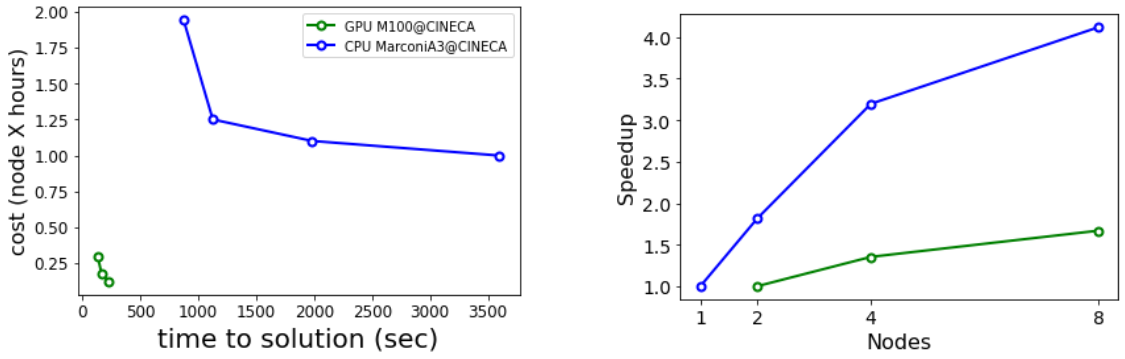

Fig. 13: Efficiency (left panel) and speedup (right) for the ZrO_2 benchmark of cp.x.

This benchmark targets the Gram-Schmidt (GS) orthogonalization routine used in cp.x. The GS is used in the initialization phase and in other simulation types implemented in cp.x. The computational cost of the GS scales:

1. with the square of the number of bands times the number of plane waves for what concerns the scalar product between bands;
2. for non-norm-conserving pseudopotentials, the augmentation part of the scalar products produces an additional cost proportional to the number of bands times the number of projectors of the pseudopotentials.

In this benchmark (Fig. 13) the calculation is composed by the wave function initialization plus 10 steps of molecular dynamics (MD). The simulated system is constituted by a slab of ZrO_2. The system has 792 atoms and 3168 bands. Because of these dimensions and of the very short duration of the MD phase, the time to solution is mainly determined by the cost of the initialization phase and within this by the GS part.

The MPI+OpenMP parallelization works by distributing the bands in different groups (band groups) and, within each band-group, by distributing coefficients and wave-function operations among the MPI ranks. For what concerns the computation of the augmentation part, within each band group, the projectors are also distributed among the ranks. OpenMP is mainly used within each rank to vectorize wave-function operations.

The *accelerated* version for heterogeneous machines maintains the same setup of the MPI groups with all the OpenMP regions directly translated to OpenACC. To improve the performance with a smaller number of MPI ranks we also distribute the computation of the augmentation part over OpenACC *gangs.* This allows for optimal performance already with the minimal number of nodes needed to meet the large device memory requirements of this test.

The execution times are shown in Tab. 7: it is evident that the openACC parallelization significantly improves the performance on heterogeneous machines. In terms of time-to-solution, the GPU version is faster. The effective usage of the computational resources --in terms of node-hours-- is also

orders of magnitude better (Fig. 13) in the accelerated execution as one can clearly see in the left panel of Fig. 13.

# nodes	M100@CINECA t.t.s (secs)	MarconiA3@CINECA t.t.s (secs)
1	--	3600
2	222	1983
4	164	1126
8	133	874

Table 7: Execution times for the ZrO_2 benchmark in an heterogeneous machine (M100) and in a homogeneous HPC machine.

Further work is ongoing for reducing the amount of device memory needed and thus the minimal number of nodes needed for the calculation. This will bring an improvement on the side of efficient resource utilization. Other improvements are expected on the side of band group parallelism, improving the general speedup for both architectures.

Siesta

Summary of identified bottlenecks

Acceleration of Cholesky step in diagonalization solvers

The previous cycle of Siesta optimization within MaX saw the integration of GPU acceleration capabilities for the diagonalization solver by means of the ELPA library, both through a native Siesta interface and through the ELSI library. Significant speedups were reported for most of the phases of the operation of the solver, including transformation back and forth from the tridiagonal form. However, the Cholesky step that transforms the generalized eigenvalue problem to a standard one (needed since Siesta uses a non-orthogonal basis) was not accelerated (see Fig. 14).

The Cholesky decomposition of the overlap matrix can be kept and re-used during the scf cycle, so the cost is incurred only once per scf cycle. For single-point calculations, which typically use on the order of tens of scf steps, this is not a problem, but it is for molecular-dynamics calculations in which the converged density matrix or Hamiltonian from a previous geometry can be reused to accelerate the scf convergence.

Then, the number of sfc steps per MD step is typically small, and the Cholesky step becomes a bottleneck. We contacted the ELPA developers about this, and they agreed to put the acceleration of the Cholesky step near the top of their priority list.

Density-matrix building step from wavefunctions

When using the diagonalization solver, an extra step is needed to construct the density-matrix (DM) from the occupied wavefunctions. The DM is a central object in the Siesta implementation, containing the electronic-structure information in a convenient (and typically sparse) data structure. The sparsity of the DM is, however, an obstacle to the vectorization and acceleration in modern architectures, including GPU-accelerated ones (see Fig. 14). In contrast to the Cholesky step discussed above, the building of the DM is needed at every scf step. We are working towards defining strategies to improve the performance of this step in modern architectures. One obvious idea, trading memory for processor efficiency, needs to be analysed with care so that scalability is not compromised.

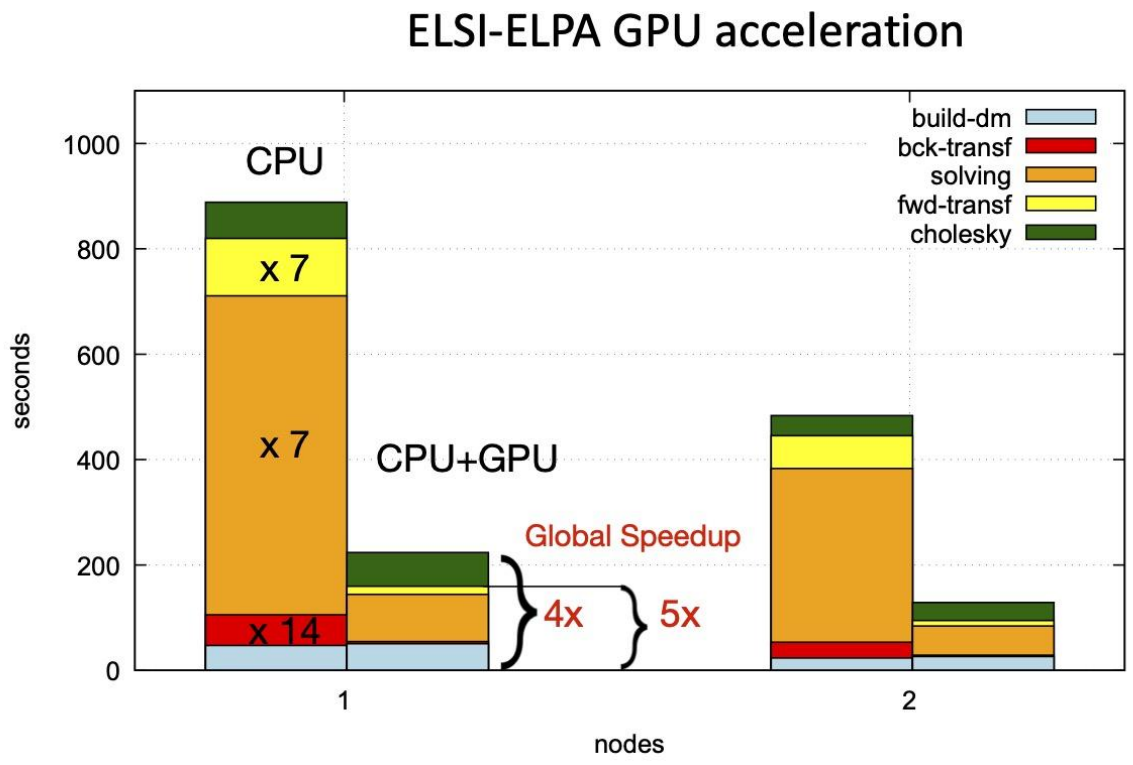

Fig. 14: Speedups obtained with the GPU acceleration of the ELPA library, as driven by the ELSI interface layer, for a SIESTA run with approximately 35000 orbitals, in Marconi 100. The relevant phases of the operation of the solver are represented by different colors.

Implications of size of Hamiltonian spectrum for FOE linear-scaling solvers

This is a well-known bottleneck that has been already reported in connection with the integration of the CheSS library for solving the electronic structure problem with the Fermi Operator Expansion (FOE) method, a linear-scaling scheme. The number of terms needed in the polynomial expansion of the Fermi-Dirac function grows linearly with the size of the Hamiltonian spectrum (note in contrast that our other massively parallel solver, PEXSI, has only a logarithmic dependency). The size of the spectrum is directly related to the cardinality of the basis set, and hence we proposed (actually as one of the algorithmic developments in WP3) to implement a basis-contraction scheme that could reduce that cardinality without compromising accuracy. We can now report that we have a first implementation of the scheme which works as expected, and improves on previous efforts in the literature by being able to mix primitive orbitals with different angular momentum. The new scheme provides a boost in efficiency across the board (not limited to the linear-scaling FOE solver), and at the same time, through the coefficients of the "hybridized" atomic-based orbitals, offers extra chemical information that we are now beginning to explore. Further details are given in the relevant section of the WP3 deliverable.

Pre-conditioning of initial step in OMM linear-scaling solver

Another WP3-related implementation is a refactoring of the traditional linear-scaling solver in Siesta, based on the Orbital Minimization Method (OMM). The refactoring is discussed in more detail in the WP3 deliverable. Here we just note that the solver has gained performance and flexibility of operation (through the abstraction of the needed matrix handling and the dispatch to appropriate backends). A remaining bottleneck is the initial convergence of the algorithm towards the basin of the orbital coefficients in the absence of a proper pre-conditioning. Typically, over a thousand conjugate-gradient steps are needed for the minimization of the modified energy functional in the first SCF step (see Fig. 15). This problem is not present in the dense form of the scheme, but that one scales as the cube of system size. We are working towards the treatment of this bottleneck. A starting diagonalization could obviously be a cure, but it would not scale appropriately.

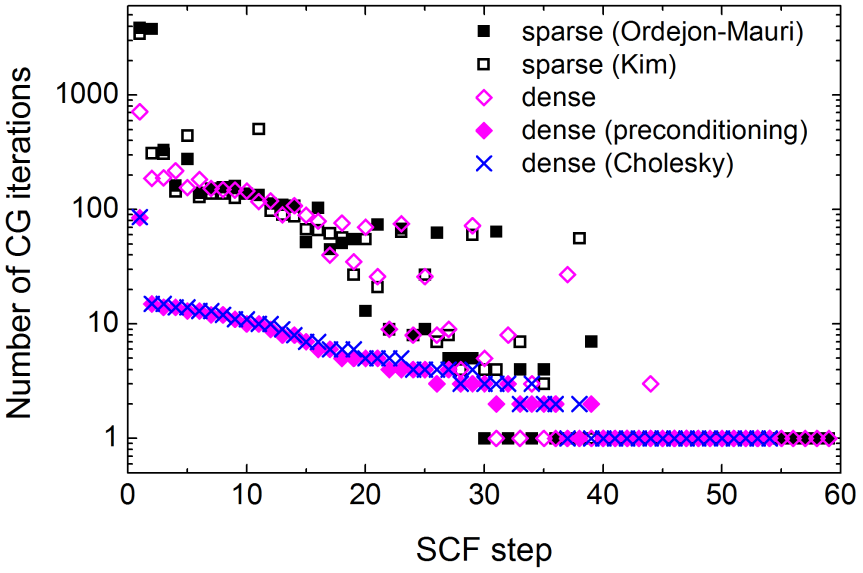

Fig. 15: Number of conjugate-gradient (CG) iterations as a function of the index of the self-consistent-field (SCF) step for the calculation of the ground state of 12x12 supercell (288 atoms) of boron nitride from scratch using the OMM scheme. Preconditioning or Cholesky transformation (not available in the sparse form of the scheme) is essential for the acceleration of the convergence in the first steps.

IO: tailored parallel distributions to avoid reordering and optimize disk access

Even though the I/O workload of SIESTA is typically moderate, it can become a bottleneck and limit the scalability under certain circumstances. This applies in particular to the writing of the sparse matrices, such as the density matrix, the Hamiltonian or the overlap matrix. The density-matrix and Hamiltonian might need to be written to disk at every SCF step for checkpointing purposes. Other data structures, such as charge densities, potentials, and wavefunctions, are typically needed on disk only at the end of the run for post-processing purposes. However, new developments now in the pipeline, such as on-the-fly computation of the thermal flux, might need the transfer of sizable datasets.

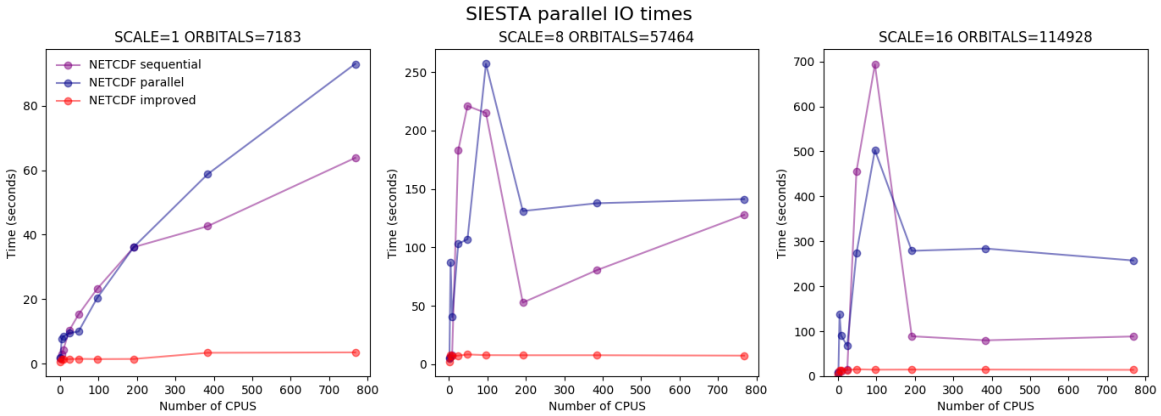

Fig. 16: Walltime for the writing to disk of the Siesta density matrix, for three different system sizes (with approximately 7000, 57000, and 115000 orbitals, respectively), and using three different I/O strategies.

SIESTA contains an interface to the parallel version of the NetCDF library that enables in principle the distribution of I/O over all the cores used by the simulation. We have performed detailed benchmarks of the I/O using various approaches, and found that the currently used approach does not perform well. As can be seen from the following figure, the parallel NetCDF performance decreases when the number of cores is increased. When performing the write operations in sequential form, i.e. allowing only one single core to write to disk, the performance slightly improves in most cases, but remains unsatisfying. This hints at the need to limit the number of disk accesses, so we implemented a new scheme that trades memory usage for transfer time. MPI processes send their data to the I/O master process, and the latter employs staging buffers to reorder the data so that very few disk operations are needed. As can be seen from Fig. 16, the new version now scales much better, and the writing time is almost independent of the number of cores used. We are currently studying intermediate solutions, such as writing in parallel but using only a subset of the processes, and employing intermediate parallel distributions of the data so that the reordering buffers are not needed.

Yambo

The Yambo code implements extensive functionality for memory and time profiling of the various sections of the code, that can be enabled at compile time. The system used as benchmark is a defective 2×2×3 TiO2 rutile bulk supercell with an interstitial H impurity (72+1 atoms), the same system used also in the two previous MaX phase-2 deliverables on code profiling and bottleneck identification (D4.2 and D4.3). Henceforth it will be referred to as the rutile-H benchmark. Also, the chosen target architecture is the same (Marconi-100 at CINECA), and this gives us the opportunity to have a very good comparison between three versions of the Yambo code. All the three versions used for this benchmark are a sort of pre-release (corresponding to v4.5, 5.0, and v5.1, respectively) and this is why they are reported as “dev” versions in the label of the figures. The development of the code was intense and aimed both at introducing new features and solving some issues. Here below will be reported only two of the main issues considered as bottleneck: the first related to the

computation of the dipole matrix elements, and the second related to the communication between the processes of the header data of the dipole matrix file.

Dipoles bottleneck

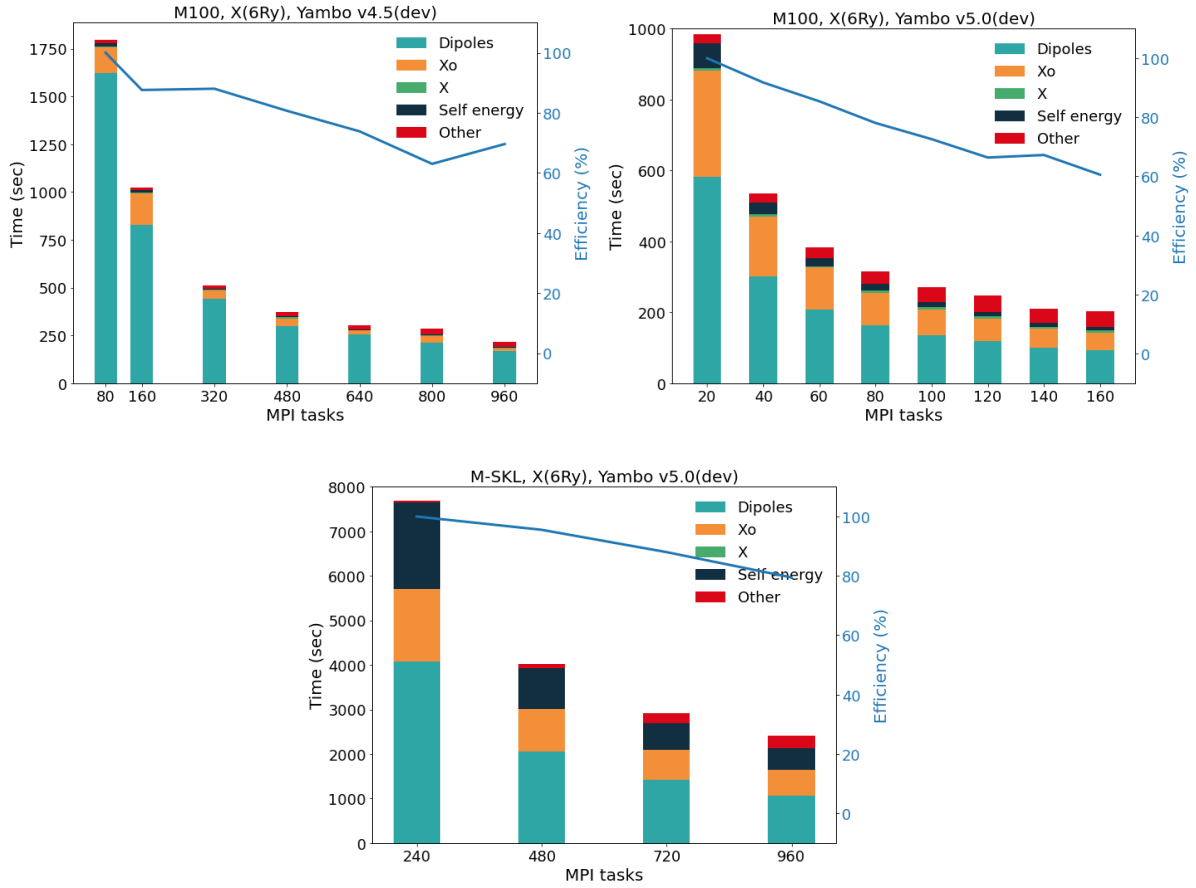

Fig. 17: Rutile-H benchmark executed on Marconi-100 at CINECA; comparison between the versions 4.5(dev) (top left) and 5.0(dev) (top right) of the Yambo code. The same benchmark was also executed on Marconi-SKL at CINECA (Intel Skylake architecture, with 48 cores per node). The Dipole bottleneck was detected and solved (see bottom plot) also for not accelerated architectures.

The profiling work done in the context of the previous deliverable (D4.2) allowed us to identify a possible bottleneck in the driver that computes the dipole matrix elements. As easily seen in the left plot of Fig. 17, the dipoles require on average 82% of the calculation time for the chosen input data. Shortly before the submission of the deliverable D4.2 we were able to re-implement the GPU porting of dipoles, obtaining a significant improvement in the timing, see the right plot of the Fig. 17. With this new implementation the dipoles require on average 52% of the calculation. The most interesting comparison is the one related to the same number of nodes or MPI tasks. Considering the calculation on 20 nodes (80 MPI tasks), before the fix on the dipoles this part of calculation spent 1623 s while after the fix it is decreased at 162.77 s with a speedup of 10. The walltime goes from 1796 s to 315 s that means an overall speedup of 5.7. For completeness, in the bottom panel of Fig. 17 we report the scaling of the same system as computed on the Marconi-A3 partition (equipped with Intel Skylake

CPUs), also showing a less pronounced role of the dipole kernel in the overall time-to-solution with respect to the initial GPU-porting without bottleneck fix.

Yambo version	MPI tasks	Dipoles	X_0	X	Self energy	Other	Walltime
4.5(dev)	80	1623.00	132.14	7.91	16.95	16.00	1796
5.0(dev)	80	162.77	91.14	31.22	18.58	11.28	315

Table 8: Rutile-H benchmark executed on Marconi-100 at CINECA. Comparison between the versions 4.5(dev) and 5.0(dev) of the Yambo code for the single run on 20 nodes (using 4 GPUs and 4 MPI tasks per node).

Asynchronous I/O

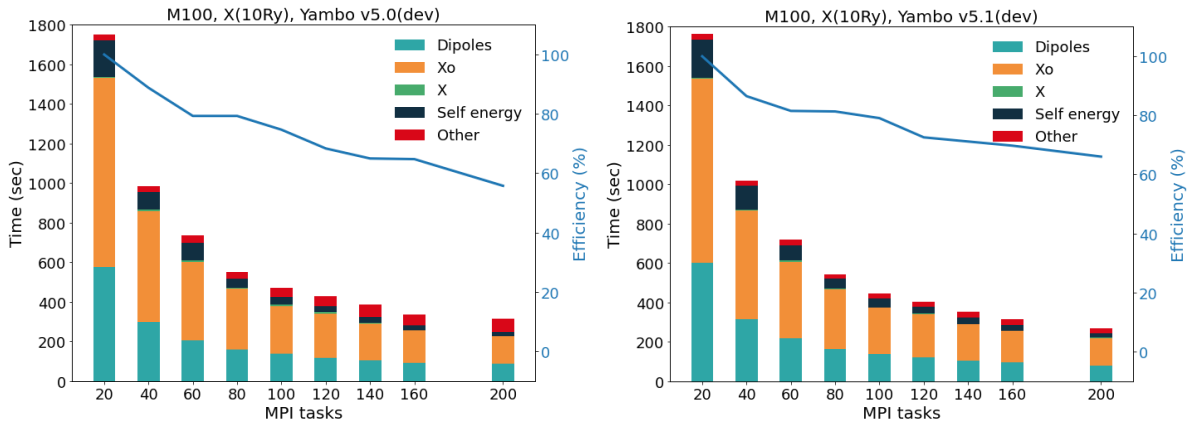

Fig. 18: Rutile-H benchmark executed on Marconi-100 at CINECA. Comparison between the versions 5.0(dev) (left) and 5.1(dev) (right) of the Yambo code.

In the last months most of the work done in the development of the Yambo code was spent in the improvement of new features and in the addition of a new backend that implements the acceleration on GPUs using OpenACC pragmas. However, the bottleneck identification process continued using the already tested strategy, which gave excellent results. In addition to the time profile written in the report file, a detailed log file is printed by some selected processes. The comparison of these two profile reports allows us to increase an already good understanding of the code profiling. In the left plot of Fig. 18 it is easy to see that the red bar (other) increases with the number of MPI tasks. With a deeper look into the Yambo log files we were able to identify that this increase is related to the communication between the processes of the header data of the dipole matrix file. The communication was a broadcast from the root process to all the other processes and this is the issue that led to less scalability. The problem was solved by introducing a mechanism that allows for an asynchronous I/O operation. After the writing of the dipoles matrix on disk, the root process continues the calculation leaving the other processes with the possibility to read the header data

directly from the file, and before starting the reading, they check for the presence of the file up to a given maximum time-limit. This solution leads to an increase of efficiency of scalability from 55.8% to 66.0% on 50 nodes (200 MPI tasks), see Fig. 18.

Conclusions

This is the last report foreseen for the profiling and benchmarking activity of MaX phase-2. This report focuses mostly on the results of the benchmarking activity and on the outcomes in terms of improved performance and usage experience for the users.

The screening during this whole period has covered a large variety of use cases. The benchmarks have been tested using large numbers of MPI ranks plus threads for multicore homogeneous systems or accelerators for heterogeneous architectures. These large amounts of computational resources were meant to approach as close as possible those that will likely be generally accessible in future exascale machines. In this sense the results of the benchmarking activity can give a first baseline of what materials scientists will be able to do on exascale machines with our codes.

The last results of the benchmarks demonstrate that the codes perform efficiently on the currently available supercomputers and their performance evolves together with the increase of available computational power. This is very promising for what the overall performance of MaX codes will be in exascale machines, but they also point out that the solution of residual scaling and efficiency issues can bring further performance improvements.

The profiling activity has been in the first part of the task oriented to the identification of code inefficiencies and was mostly done by the code developers. The focus has then passed to more technical aspects related to architectural and co-design questions. It has for this reason passed under the care of the supercomputing specialists of HPC centers.

For completeness and coherence, most of the reporting on profiling has been thus moved to the D4.6 deliverable written together with the present document. Also the energy consumption analysis, closely related to profiling, has been moved to D4.6 for the same reasons.

The next report on performance portability will give a synthetic view of these activities. The benchmarking activity will keep going in the next months. These will continue to be a valuable tool to assess the code efficiency on new systems, notably including EuroHPC pre-exascale machines as soon as available. For this reason we are working at making the content of the benchmark repository more readable and accessible to third parties users.