# D4.3

# Second report on code profiling and bottleneck identification

Fabio Affinito, Uliana Alekseeva, Carlo Cavazzoni, Augustin Degomme, Pietro D. Delugas, Andrea Ferretti, Alberto Garcia, Anton Kozhevnikov, Pablo Ordejón, and Nicola Spallanzani

| | |
|---|---|
| Due date of deliverable: | 31/05/2020 |
| Actual submission date: | 31/05/2020 |
| Final version: | 31/05/2020 |
| 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) / 30/11/2021 (month 36) |
| Website: | www.max-centre.eu |
| Deliverable No.: | D4.3 |

Authors: F. Affinito, U. Alekseeva, C. Cavazzoni, A. Degomme, P. D. Delugas, A. Ferretti, A. Garcia, A. Kozhevnikov, P. Ordejón, and N. Spallanzani.

To be cited as: F. Affinito et al. (2020): Second report on code profiling and bottleneck identification. Deliverable D4.3 of the H2020 project MAX (final version as of 31/05/2020). EC grant agreement no: 824143, CINECA, Casalecchio di Reno (BO), Italy.

## Disclaimer:

# D4.3 Second report on code profiling and bottleneck identification

## Content

# 1 Executive Summary

In the present deliverable, we report the progress made on the benchmarking of the MAX flagship codes, with reference to the test cases defined in the D4.2 document. Importantly, part of the benchmarks run from M6 to M18 (May 2019 - May 2020) were already reported in D1.2, together with the release of the MaX codes in November 2019. In the present document we therefore report the newest data, mostly harvested in a benchmarking campaign held during spring 2020.

Notably, in the month of March the production of Marconi100, a >30 PFlops cluster based on IBM Power9 + nVIDIA V100 cards, has started at Cineca. This gave us the unique opportunity to test the GPU porting of MaX codes at scale, especially during the setup and pre-production period of the machine. In this deliverable we report the early results from the benchmarks on this machine which allowed us to explore code behaviours in a very GPU-unbalanced architecture, even more relevant in view of the expected architecture of the EuroHPC pre-exascale machines to be deployed in early 2021.

This campaign allowed us to find new bottlenecks and to target new development work. A number of the early identified problems have already been addressed and we were eventually able to run massively parallel calculations using MaX codes (e.g. a ~20 PFlops single run of Yambo on 600 nodes out of 980 of Marconi100, to name one).

Concerning Quantum ESPRESSO (QE), the GPU port was extensively checked, also on large scale systems. Results are very promising and helped us to identify memory footprint bottlenecks, especially during diagonalization, furthermore stressing the need for GPU-aware distributed linear algebra primitives. The Car-Parrinello kernel of QE was also recently ported to GPUs and benchmarked at scale with very interesting results.
Yambo was ported on Marconi100 and turned out to be in excellent shape for what concerns the GPU port, except for a performance loss due to the dipole kernel. The benchmark data allowed us to address it and to propose a solution. Even more than in the QE case, the inclusion of GPU-aware distributed linear algebra libraries aiming at controlling memory usage has been found to be quite critical.
FLEUR continued its work on the JURECA cluster at Juelich, especially in the direction of improving the load-balancing of the matrix setup. A new exploitation of the k-point parallelism for large unit cells is discussed and, finally, the case for performance fluctuations is reported.
BigDFT reports the development for the execution of calculations inside AiiDA. In addition it discusses in depth the results coming from the development of *libconv*, a separate library used for the calculation of convolution elements which permits, using code-generation with a metaprogramming approach, to target many different underlying computer architectures.

CP2K reports on the results coming from the adoption of the COSMA library. These results look quite good, in particular for cRPA calculations.
SIESTA reports on the substantial speedups that can be achieved by using recent GPU-enabled versions of the ELPA library (directly and through the ELSI solver interface library). The SIESTA section also shows that the PEXSI method (not based on diagonalization) still offers the best scaling and massively-parallelization opportunities.
To finish, we report a proof-of-concept of the utilisation of AiiDA as a benchmarking tool, discussing pros and cons in comparison with JUBE, another popular tool for benchmarks and analysis of performances.

# 2 Introduction

In this deliverable we show the benchmarks of the MAX flagship codes (Quantum ESPRESSO, Yambo, FLEUR, BigDFT, CP2K, and SIESTA). In the previous D4.2 deliverable we set up a list of test cases to which we will make reference in this work.

Many benchmarks reported in D4.2 were performed on the CINECA Intel KNL system (Marconi), which has been recently decommissioned. For this reason, some of the comparisons have been reported with reference to other Intel x86 architectures (for example Intel Skylake). This is the case of Quantum ESPRESSO and Yambo, whose benchmarks have been performed in CINECA.

Some of the reported benchmarks were able to run on the new Cineca Marconi100 cluster, a machine based on Power9+NVIDIA V100, which permits to highlight new bottlenecks in a very GPU-focused architecture. Finally, we report a proof-of-concept of utilization of AiiDA as a benchmarking tool.

# 3 Update and progress on code performance

## 3.1 Quantum ESPRESSO

We present here an update of our benchmarks of the Quantum ESPRESSO codes. One first change with respect to the previous set of benchmarks regards the fact that we needed to change the reference machine where these tests are executed.

The KNL partition of the Marconi cluster of CINECA is no longer available and for this reason the new reference calculation for MPI+OpenMP machines is now the SKL partition of the Marconi cluster of CINECA. It has thus been possible to increase the size for the benchmark reference system for large size computations using a very large system recently studied in WP6 as a demonstrator.

Benchmarks for heterogeneous architectures based on GPGPUs have been instead executed on the recently deployed Marconi100 cluster at CINECA.

### 3.1.1 PW: large test case 1: The SARS-COVID19 protein

The new benchmark reference system for the large computation test case is a monomer of the main protease ($M^{pro}$) of the SARS CoV-2 virus recently used in a demonstrator case of WP6.

The FFT 3D grid is (360,576,540) with 8783 atoms and 16746 bands. The size of the system is challenging for the computational load as well as for the memory footprint estimated in a net requirement of more than 10000 GB. The total execution time for 4 SCF cycles and the contribution from the most significant kernels are reported below in Tab. 1, while the averaged times per step are plotted in Fig. 1 .

| #TASKS | **192** | **384** | **768** |
|---|---|---|---|
| NDIAG | **169** | **361** | **729** |
| total time | 5280 | 3083 | 2253 |
| init_run | 448 | 265 | 193 |
| h_psi + s_psi | 1986 | 1120 | 650 |
| rdiaghg | 1298 | 739 | 600 |
| sum_band | 356 | 210 | 142 |

**Table 1**: Main clocks for the SARS-CoV test case as a function of the number of MPI tasks and the number of tasks used for parallel linear algebra (NDIAG) . Calculations run on the Marconi-SKL partition at CINECA.

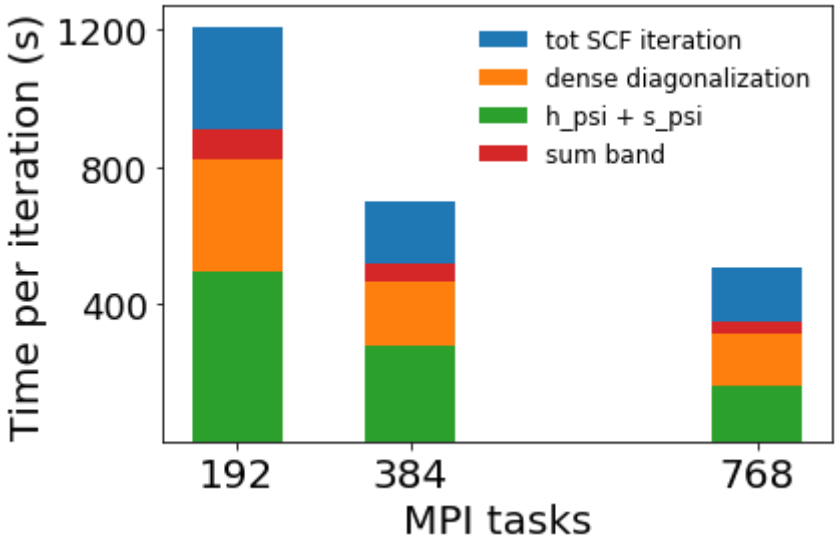


**Figure 1:** Averaged time per iteration of the SarvCovFrag system on the Marconi-SKL partition of CINECA. Each MPI task runs with 8 OMP threads.

These results confirm what was evidenced in the previous report. For large size systems, as soon as enough MPI tasks are used, the dense parallelization contribution dominates on other contributions. Improvements on this aspect may come only from the adoption of more effective linear algebra libraries or of alternative algorithms. Work on the algorithm is ongoing (at an experimental stage) and can not yet be tested on systems of such size.

### 3.1.2 PW: The CNTPor test case

The CNTPor case presented in the previous set of benchmarks (D4.2) shows how the poor scalability of the distributed parallelization becomes dominant as soon as the number of MPI tasks becomes comparable with the size of the FFT grid. In order to assess the developments that have been explored to mitigate or bypass this bottleneck, we update the benchmarks on the CNTPor case with computation done on the SKL and M100 partitions of the Marconi Cluster of CINECA.

The benchmark has been run on the Marconi SKL using 64 nodes with 3072 MPI tasks. We checked the optimal distribution of these MPI tasks using band parallelization. As shown in Fig. 2, the optimal distribution is reached when 4 band groups are used. The band parallelization is still inefficient in reducing the time spent in dense diagonalization while the distribution of the h_psi and s_psi calls among the band groups results in an effective scaling of the time spent on these routines.

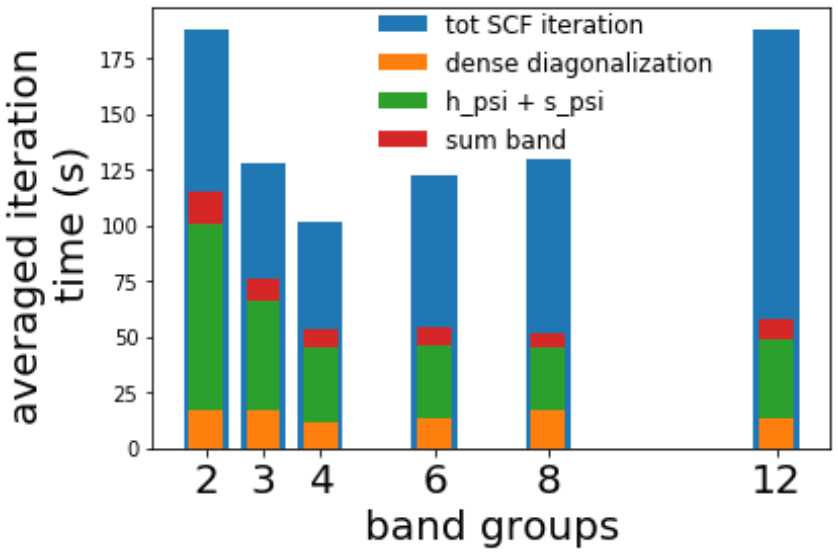


**Figure 2:** Comparison of averaged time per iteration for the CNTPor case run on the SKL partition of Marconi@CINECA using 3072 MPI tasks.

We have also run the same test case in the M100 partition of Marconi@CINECA. In this case, the main bottleneck is given by the requested memory that the program needs to allocate on the device memory. In fact, the device memory needs to be distributed on at least 104 VOLTA cards, thus requesting the usage of at least 26 M100 nodes. As shown in figure Fig. 3, once this large number of devices is reached, the scaling is already at saturation and the further increase in the number of used devices produces an impairment of the performance. This is mostly due to significant communication overhead, while actual compute time in the device remains almost unchanged with the variation of the number of devices used.

In the future development we aim at improving these benchmarks on two main points:

- The advent of efficient distributed linear algebra libraries for GPUs should provide an improvement of the performance and a reduction of the memory footprint.
- Improvements in the band group parallelism, mostly introducing algorithmic developments that should allow the code to eliminate or reduce the dense diagonalization of large matrices (as currently needed by the Davidson algorithm).
- For the GPU case, reducing the size of device memory with a better management of scratch spaces and device allocated arrays.

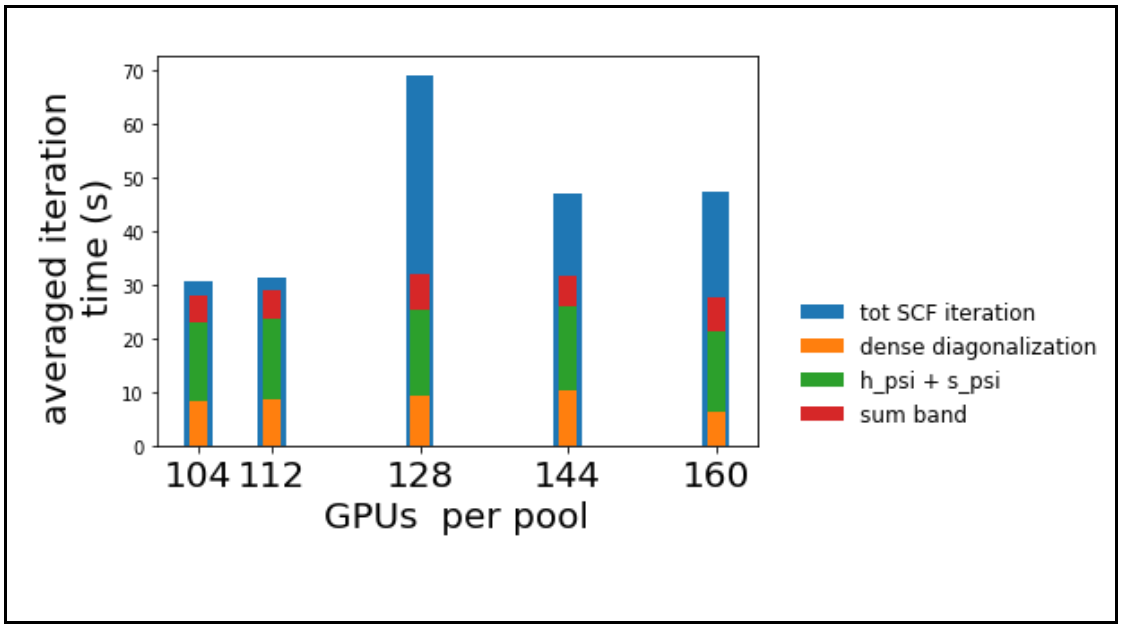


**Figure 3:** Average time of SCF iteration in the CNTpor system on the M100 Cluster in CINECA. The large number of GPUs used is due to the necessity to distribute data on a large number of devices. Main contributions from accelerated parts of the algorithm are also plotted. These parts have similar averaged times for all setups. Increasing the number of MPI tasks a significant overhead is paid.

### 3.1.3 PW: Medium test case: Graphene on Ir slab.

This system - which is referred to in the following as GRIR686 - is a medium sized test case. It is the computation of a few Ir atoms adsorbed on a Graphene sheet. The system counts 686 atoms, 3100 bands, 4 k-points, and is run with a spin-polarized GGA-PBE exchange-correlation functional. It is thus possible to use pool parallelism up to 8 pools (4 k-points times 2 spin channels). In this benchmarking campaign it has been run with a wave function cutoff of 30 Ry, requiring a FFT grid of {180, 180, 216}. With such setup and using 2 pool parallelization, it requires a total host RAM of 556 GB. The needed device memory is not yet automatically estimated by the program: with different tests, we have determined that it is necessary to distribute the data of each pool on at least 20 Volta cards. This is due to the need to leave enough memory on the device to perform dense diagonalizations on the iterative space within the Davidson algorithm (matrices 6200X6200).

We have performed tests using different numbers of GPU and pool parallelization (1 or 2). Results are reported in the Fig.4. For an indicative comparison, when running the same workload on the SKL cluster, each pool must be distributed on 16 nodes. The total time on SKL using 2 pools is 1089 seconds.

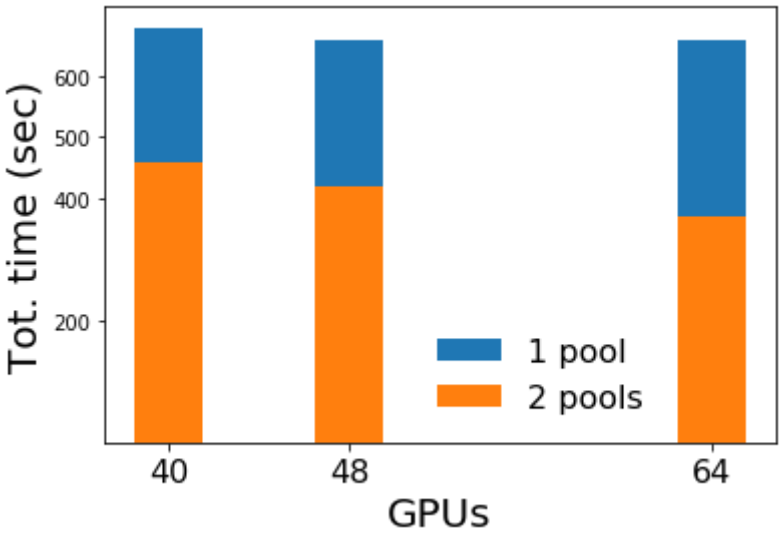


**Figure 4:** Total time spent on the SCF fraction of the code. For profiling reasons the code performs only 4 SCF steps.

To have a clearer insight on the performances of the code, we report in Fig. 5 the average time taken by each single SCF iteration in a single pool. The plot shows how the performance is already close to the optimal one at 20 GPUs per pool. The dense diagonalization contribution is performed by one device and thus does not change, while the h_psi and residual parts both show only a small improvement increasing the number of GPUs per pool.

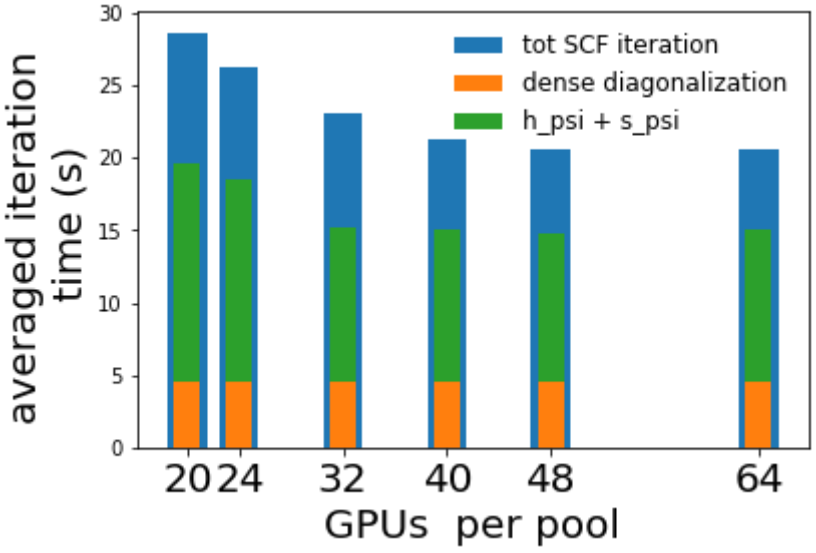


**Figure 5**: Averaged time per iteration depending on the number of used GPUs. The contribution of total time coming from dense diagonalization and

Hamiltonian-vector operations are reported. Dense diagonalization is performed by one GPU and thus does not change. h_psi part is already optimal at 20 GPUs and reaches best performance for 40 GPUs per pool.

**Considerations on the test case:**

- The memory requirements of dense diagonalization represents an important bottleneck, as they create a significant imbalance in the memory used by the devices, increasing the minimal number of devices required for medium and large size cases. We are confident that the advent of efficient libraries for distributed linear algebra and diagonalization on GPUs will make it possible to have a more uniform device memory distribution and to reduce the number of needed GPUs allowing for a more efficient usage.
- The distribution of plane waves on more devices does not increase the performance of the code because the performance is already close to saturation with the minimal number of GPUs for which it is possible to run the calculation.

### 3.1.4 CP bottleneck mitigation

In the use case ZrO2, we found a bottleneck due to "updatc" subroutine. The bottleneck was indeed more general, and related to the parallelization of the update cycle of the augmentation component of the wave functions. Among the two layers of parallelization that could have been exploited for this subroutine, namely linear algebra block-like parallelization and band parallelization, only the former was used. We then implemented the missing band parallelization, and significantly reduced the weight of the updatc subroutine over the whole time to solution, from 15% down to 7%, as can be seen comparing plots in Fig.6 and Fig.7 which report the timings of KNL runs (from D4.2 deliverable) and of the new runs SKL and K80, performed for this deliverable. Note that KNL refers to Marconi-KNL partition with Intel Xeon Phi processors, SKL refers to Marconi-SKL partition with Intel Xeon Skylake processors, and K80 refers to Galileo Tier-1 cluster using nodes with K80 NVIDIA GPUs.

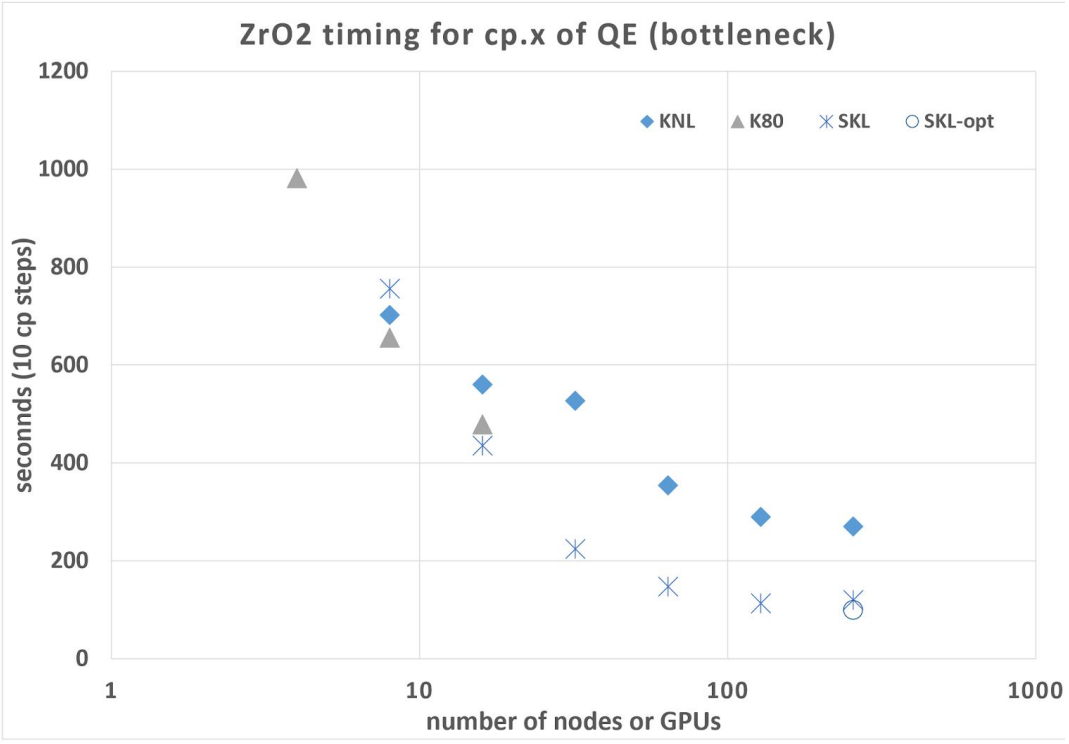


**Figure 6:** Previous ZrO2 benchmark from D4.2, to be compared with Fig. 7.

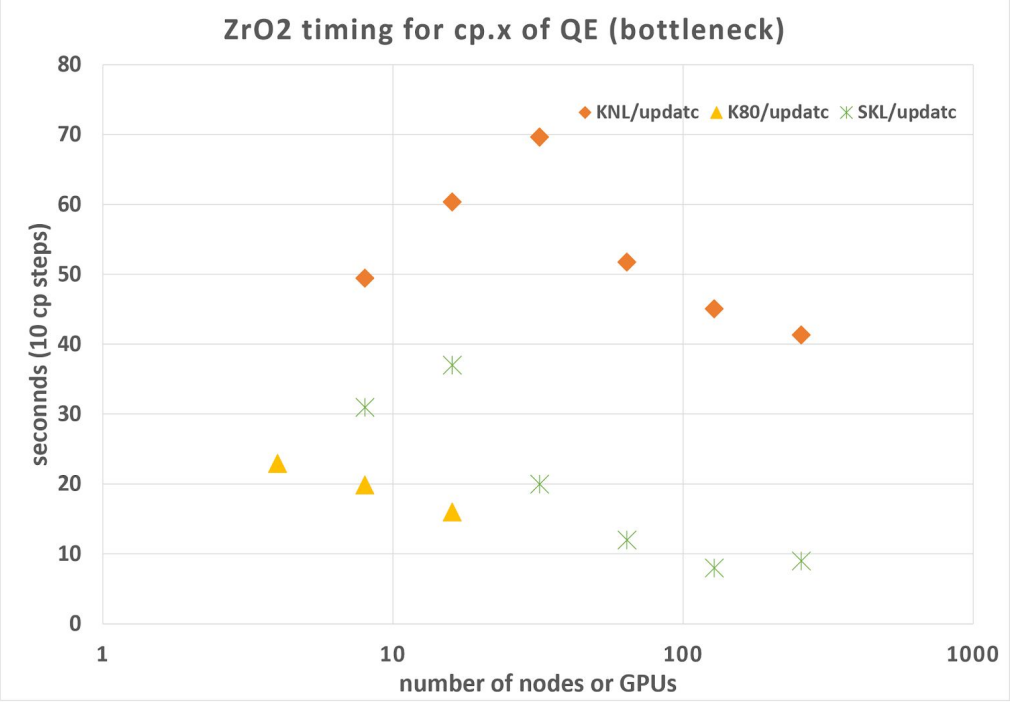


**Figure 7:** Timings of cp.x runs for the ZR02 benchmark case after the changes with execution times significantly reduced.

Here a few more things need to be highlighted. First of all it was not possible to run on the same Marconi-KNL partition, since this partition has been replaced by the Marconi-100 GPU accelerated partition. Nevertheless, the overall performance of the KNL and SKL nodes are about the same. In fact in Figure 6 and 7 we reported the number of nodes in the ascissa, and not the number of cores, since the two types of cores are instead not comparable (note that when running on 8 nodes the performance of the two architectures is almost the same). Secondly, we take advantage of the new GPU enabled version of the CP kernel (described in deliverable D4.4 “First report on co-design actions”), to compare the results obtained using a whole non accelerated node and a single GPU (we think this is the most fair metric, since the nominal performance of a K80 card is similar to the one of a KNL or SKL node).

## 3.2 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. These functionalities were already used with very good results in combination with a

scalability test performed on Marconi-KNL, as reported in D4.2. The same approach was again used with the purpose of a more in-depth improvement of the MPI+OpenMP scalability performance, as reported in D2.1. This is the reason why we decided to continue with this strategy also for the GPU version of Yambo. The code was almost completely ported on GPU using CUDA Fortran. However, as the porting is very recent and the code was never tested on GPU at scale, we expected some parts of the code to be in need of improvement.

For the test we have considered a defective 2×2×3 TiO2 rutile bulk supercell with an interstitial H impurity (72+1 atoms). The same system was used to perform the scalability test in section 4.2.2 of D4.2. There are two reasons to use the same system. The first is that it is possible to see the speed up obtained by making use of GPUs for the same number of nodes. In Table 2, we report the walltime (in seconds) of the calculations performed for the system above mentioned . The second line of the table is related to the Marconi-KNL cluster, equipped with nodes with 68-cores Intel Xeon Phi 7250 CPU (Knights Landing) at 1.40 GHz. The third line is related to the new cluster Marconi100, equipped with nodes with 2x16 cores IBM POWER9 AC922 at 3.1 GHz and 4x NVIDIA Volta V100 GPUs per node (Nvlink 2.0, 16GB). The last line of the table shows an average speed-up of 5.7 thanks to the GPU acceleration. This is a very good result.

| # Nodes | 40 | 80 | 120 | 160 | 200 | 240 |
|---|---|---|---|---|---|---|
| M-KNL | 5724 | 3477 | 2134 | 1662 | 1379 | 1286 |
| M100 | 1025 | 510 | 371 | 304 | 285 | 215 |
| speed-up | 5.58 | 6.82 | 5.75 | 5.47 | 4.84 | 5.98 |

**Table 2**: comparison of scalability tests at the same number of nodes between two clusters with very different architectures, Marconi-KNL (M-KNL) and Marconi100 (M100) both installed at CINECA supercomputing center.

However we think that it is possible to obtain a better speed-up analyzing the parts of the code that can be improved. One of the most computationally intense parts of the GW kernel for this specific system is the calculation of the dipole matrix elements. This is the second reason why we decided to use this system for the test, and Fig. 8 shows very well that the calculations of the dipoles is the part of the code that needs an in depth analysis in order to optimise the run.

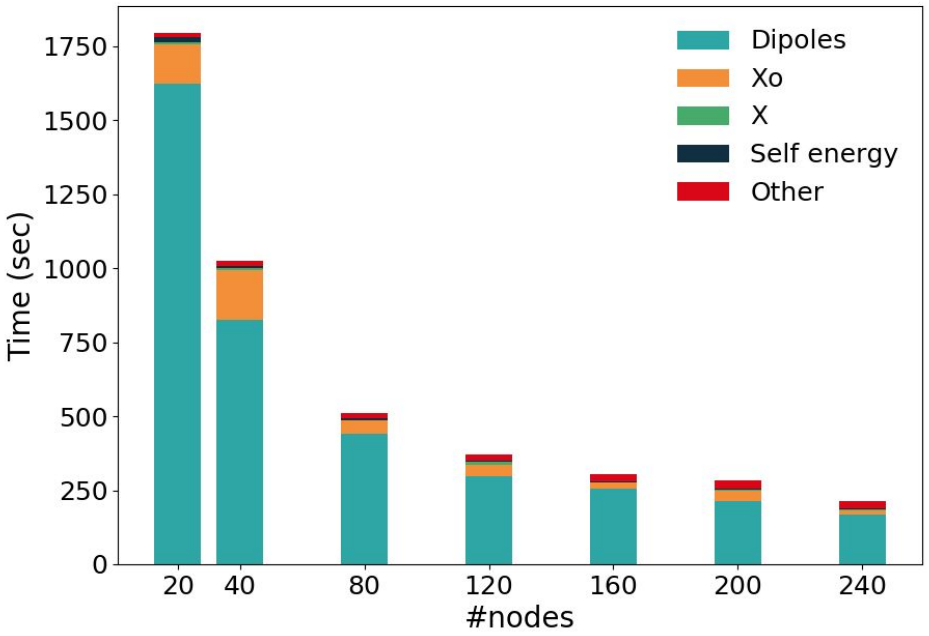


**Figure 8**: Execution times for the tests performed with the Yambo code to verify the efficiency of the GPU porting.

The times reported in the Tab.3 show that the dipoles require on average the 82% of the calculation. A preliminary check on the use of the GPU during this part of the calculation, through the use of the nvidia-smi tool, reveals a GPU usage that does not exceed 45%. All other kernels show a much larger usage (typically close to 100%). The next step we intend to carry out is a complete profiling of the calculation using the nvprof tool.

*Note added:* prior to the submission of this document we were able to re-implement the GPU porting of Dipoles, obtaining a significant improvement in the timing.

**Table 3**: Tests have been performed on Marconi100 using 4 MPI tasks per node, 32 threads per task and a 1:1 binding between MPI tasks and GPUs. Times are given in seconds.

| # Nodes | # MPI | # Threads | Dipoles | Xo | X | Σx | Σc | wall_time |
|---|---|---|---|---|---|---|---|---|
| 20 | 80 | 32 | 1623 | 132.1358 | 7.9136 | 4.1661 | 12.7836 | 1796 |
| 40 | 160 | 32 | 827 | 165.7774 | 6.3793 | 2.6552 | 6.7998 | 1025 |
| 80 | 320 | 32 | 441.6614 | 43.3924 | 2.826 | 1.844 | 4.0862 | 510 |
| 120 | 480 | 32 | 298.6063 | 38.8259 | 7.4733 | 1.6332 | 3.245 | 371 |
| 160 | 640 | 32 | 254.5834 | 20.4671 | 1.9817 | 1.4164 | 3.0007 | 304 |
| 200 | 800 | 32 | 212.9672 | 35.503 | 2.8716 | 0.6085 | 3.2236 | 285 |
| 240 | 960 | 32 | 168.9603 | 14.446 | 1.5422 | 0.5046 | 2.8768 | 215 |

## 3.3 FLEUR

This Section reports the performance improvements of the FLEUR versions MAX Release 3.1 and MAX Release 4.0.

### 3.3.1 Load balance in the Matrix Setup

As outlined in the previous deliverable D4.2 "First report on code profiling and bottleneck identification, structured plan of forward activities", the performance of the FLEUR code needed to be reevaluated after the implementation of the new data layout (Del. 4.2, par. 4.3.2). The calculations for that were done with the same test case (CuAg 256 atoms) on the same machine (CLAIX 2016, Intel Broadwell E5-2650v4, 24 cores/node, peak performance 35 GFlops/core). The data (Tab.4) show significant improvements: dashing +124% for the matrix setup part, which corresponds to the total performance increase of 42%. As mentioned in the previous deliverable, such a large improvement in the matrix setup part became possible due to the utilisation of BLAS kernels.

| | Matrix setup | Diagonalization | New charge | Total |
|---|---|---|---|---|
| CPI | 0.48 | 0.37 | 0.44 | 0.42 |
| Performance, GFlops | 17.3 (+124%) | 23.3 (+12%) | 6.4 (+11%) | 17.2 (+42%) |

**Table 4**: Performance counters measurements done by LIKWID, average values per core. Code: FLEUR MAX Release 3.1, hardware: CLAIX 2016, one node (24 cores). Test case: CuAg 256 atoms. The percentage shows the improvement in performance due to the new data layout.

To verify that the load balance in the matrix setup part was not impaired by the introduction of the new data layout, the trace of the parallel execution (8 nodes, 4 MPI processes per node spawned to 6 OpenMP threads each) was collected (Fig. 9).

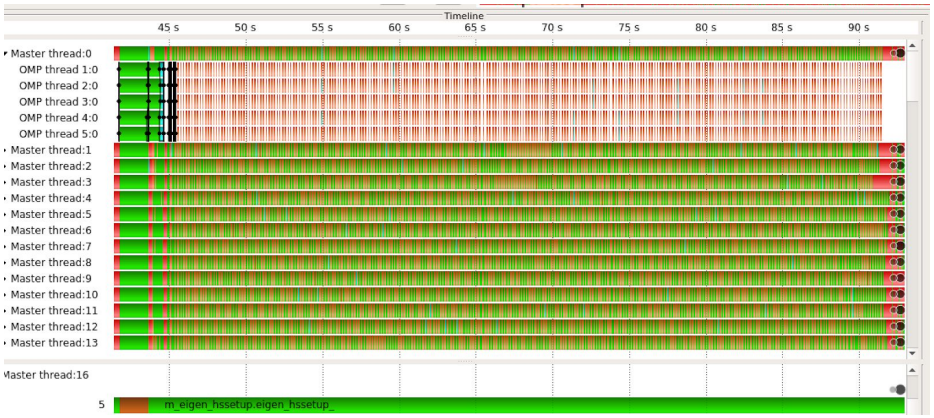


**Figure 9**: Trace of the parallel execution of the FLEUR code (MAX Release 3.1), matrix setup region. Only 13 of 32 MPI processes are shown, the first MPI process is shown with its worker OpenMP threads.

### 3.3.2 Spherical Matrix Setup

One of the most computationally intensive parts of the code, the matrix setup, itself consists of several algorithmically very different subroutines. The improvements presented in the previous section were achieved mostly by the utilization of the BLAS calls in the so-called non-spherical part. The spherical part can not be represented in this way, but a careful restructuring of the main loop and several internal arrays increased the reusage of the data in the cache and allowed the compiler to apply vectorization efficiently. From the time measurements (Tab. 5) it can be seen that the significance of this improvement grows with the system size. The last two test cases (TiO2 big and TiO2 huge) are the scientific use cases for the profiling. These measurements are done on the JURECA Cluster at the Forschungszentrum Jülich (Intel Haswell E5-2680 v3, 24 cores/node).

| | # atoms | # nodes | Sph, s | Sph_opt, s | Speedup |
|---|---|---|---|---|---|
| CuAg | 256 | 1<br>2<br>4<br>8 | 172.12<br>76.81<br>41.83<br>21.71 | 89.06<br>43.78<br>22.2<br>10.45 | 1.9<br>1.8<br>1.9<br>2.1 |
| GaAs | 512 | 8<br>16<br>32<br>64 | 335.23<br>168.11<br>83.24<br>39.65 | 159.82<br>78.75<br>38.06<br>18.45 | 2.1<br>2.1<br>2.2<br>2.1 |

| | | | | | |
|---|---|---|---|---|---|
| TiO2 big | 1078 | 32<br>64<br>128 | 369.71<br>174.14<br>84.63 | 157.21<br>75.51<br>35.59 | 2.4<br>2.3<br>2.4 |
| TiO2 huge | 2156 | 256 | 767.78 | 242.49 | 3.2 |

**Table 5**: Execution time measurements in seconds of the spherical matrix setup subroutine, done for four test cases on the JURECA Cluster (Forschungszentrum Jülich). The number of atoms in each test case are given in the 2nd column. The optimized version (Sph_opt, MAX Release 4.0) is compared with the not optimized one (Sph, MAX Release 3.1), the corresponding speedups are given in the last column.

### 3.3.3 K-scaling for large unit cells

The FLEUR code has two levels of MPI parallelization: i) over k-points and ii) over the eigenvalue problem. Since consideration of $n$ k-points leads to $n$ independable eigenvalues problems, the parallelization over the k-points shows almost ideal scaling. However, the necessity to calculate many k-points is usually there only if the unit cell is quite small, that is why this behaviour was not so far confirmed with simulations of large systems. One of our scientific use cases, SrTiO3, needs to be simulated with several k-points due to a very flat geometry. The calculations with 1, 2, and 4 k-points on 256, 512, and 1024 nodes (SuperMUC-NG, Intel Skylake Xeon Platinum 8174, 48 cores/node) showed indeed a ideal scaling behavior: the execution time was the same (with the deviation within the 2%, which is the same as a statistical deviation of the repeated identical calculations).

### 3.3.4 Performance fluctuations

We reported in D4.2 that significant performance fluctuations (2x-4x) were observed while running large benchmarks (> 1000 atoms) on the CLAIX supercomputer. At the time it was not clear what caused it. We reported these measurements to the CLAIX administrators and they found some hardware malfunctions. The performance fluctuations have reduced considerably after those malfunctions were eliminated. We also performed a considerable amount (19) of numerically identical calculations on 1024 nodes of SuperMUC-NG and all execution times were the same (with a deviation of 2%), hence we assumed that this was not a flaw or bottleneck of the code.

## 3.4 BigDFT

### 3.4.1 AiiDA plugin

A first version of the aiida-bigdft plugin has been released, providing support for simple BigDFT calculator and log file parsing. Integration of AiiDa calculators in BigDFT notebooks and Datasets has also been added, to allow launching of computation on a remote HPC system from an existing notebook with no changes.

A second version will soon be released, supporting common AiiDa workflows in the plugin, as part of WP5.

### 3.4.2 Performance prediction of the libconv library

Convolutions in BigDFT have already been identified as a main point of focus for performance improvement in the near future. They represent a large part of most computations, and have been finely tuned years ago, on outdated architecture. SSE hard coded instructions can be efficiently converted by compilers in AVX instructions, but they fail to scale for larger vector sizes and won't reach near peak performance on most systems anymore. In order to reduce the burden on developers for their code optimization and support new architectures easily, a new solution was selected: a separate library for convolutions, called libconv, with code generation through meta-programming, and auto-tuning for performance. Convolutions written with BOAST DSL are generated with various optimization (vectorization using various vector sizes and instruction sets, loop unrolling, dimension reordering, ..), benchmarked and the fastest for each case selected for use in the final library.

In parallel, one goal of BigDFT is to provide users with advice on which method is better suited for their needs. Linear scaling BigDFT usually provides best performance for larger systems with more nodes involved, but this is not true for all input sizes and HPC systems. Having a decision tool to help users run the most optimized input set for their need would reduce the amount of computation hours needed to get results, enhancing efficiency by a huge factor.

In this regard, BigDFT has already been simulated accurately using the SimGrid's SMPI framework[1], which simulates the behavior of a MPI library and the networking part of an HPC system, in order to diagnose potential issues and estimate runtime on various platforms. But by default the time taken for a simulation is the total computing time of a process multiplied by the number of simulated MPI processes, as each

[1] [SMPI] Augustin Degomme, Arnaud Legrand, Georges Markomanolis, Martin Quinson, Mark Stillwell, et al.. Simulating MPI applications: the SMPI approach. IEEE Transactions on Parallel and Distributed Systems, Institute of Electrical and Electronics Engineers, 2017, 28 (8), pp.14. ⟨10.1109/TPDS.2017.2669305⟩. ⟨hal-01415484v2⟩

computation kernel is executed sequentially on a single node. Same thing is true for memory needs, as all data has to be allocated on a single node. This means that we need to drastically reduce these two costs to give an accurate and fast estimation of the computation time for a given input set to our users, without changing large parts of code inside BigDFT.

For memory, BigDFT already wraps memory allocations using custom allocators. SMPI provides "shared" allocators, which by default returns a single memory block (by default of size 1MB) and maps it multiple times to match the asked size. This means that these multiple calls from multiple simulated MPI processes of any size will actually use a tiny amount of memory, and loop over it without noticing. This can only be done when data itself is not relevant, for instance when simulating the cost of a single iteration of a process (no convergence needed), and not for control data.

A simple change has been implemented in BigDFT allocators to add the option to use these shared allocators. As BigDFT futile library - which handles the allocators - is heavily dictionary-oriented, this actually meant adding a single element to a descriptor dictionary when calling the allocator, allowing for painless switching to SimGrid's allocators for selected calls, only when available.

For time intensive computing kernels, the typical solution in SMPI is to benchmark each computing kernel multiple times during the computation, and when the result is stable enough, to skip computation for the next calls/iterations and simply inject this stable time in the simulation engine. This works nicely for C codes, but is not suited (heavy use of macros) for Fortran codes, and it can be intrusive, as kernels have to be carefully selected and may need to be reworked to be wrapped correctly. Furthermore, injecting a constant time for each kernel call has been proven not to be accurate enough in some cases, as variability can be important on some systems, resulting in a potential incoherence between simulated and real behaviors of application[2].

The development of libconv allows us to develop a new technique. When kernels are generated, their computational cost is also estimated and provided to the user through helper functions. This cost can also be compared to the real computation time, corrected to account for noise and the particular speedup of some kernels, and injected into SimGrid, skipping computation altogether. For this we implemented through the meta-programming DSL of BOAST a switch that can be activated at runtime *via* the environment, to provide three different execution modes :

[2] [CLUSTER]Tom Cornebize, Arnaud Legrand, Franz Heinrich. Fast and Faithful Performance Prediction of MPI Applications: the HPL Case Study. 2019 IEEE International Conference on Cluster Computing (CLUSTER), Sep 2019, Albuquerque, United States. ff10.1109/CLUSTER.2019.8891011ff. ffhal02096571v3

- default mode, with normal execution of the underlying kernels (multiple kernels may be used for each convolution call)

- benchmarking mode : each call is timed and the results, with all parameters used for the call to the kernel, are written in a CSV file. This data can then be processed to evaluate the behavior of kernels, through R scripts for now (Fig. 10). This script outputs for each kernel (which can actually hide several implementations) regression coefficients between the estimated cost of computation, and the actual times, with noise estimation. This can be performed on a single process run.

- injection mode : instead of executing kernels, the previous coefficient file is read and used to compute the time to inject for each kernel call, potentially accounting for noise (not needed in most cases).

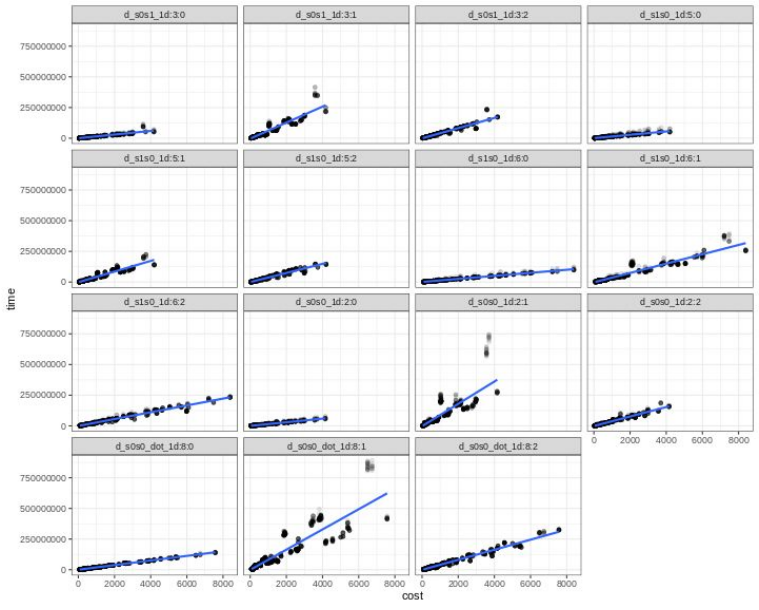


**Figure 10:** Behavioral study of execution time (seconds) vs estimated theoretical cost (MFlops) of 15 different convolution operations from libconv on a real platform (single node from galileo at CINECA) when randomizing several input parameters. For most of them noise seems negligible and linear approximation seems sufficient for accurate simulation. Two of them show slight variability, probably due to the switch of underlying implementations depending on the input parameters, and would need noise injection to be more accurately simulated.

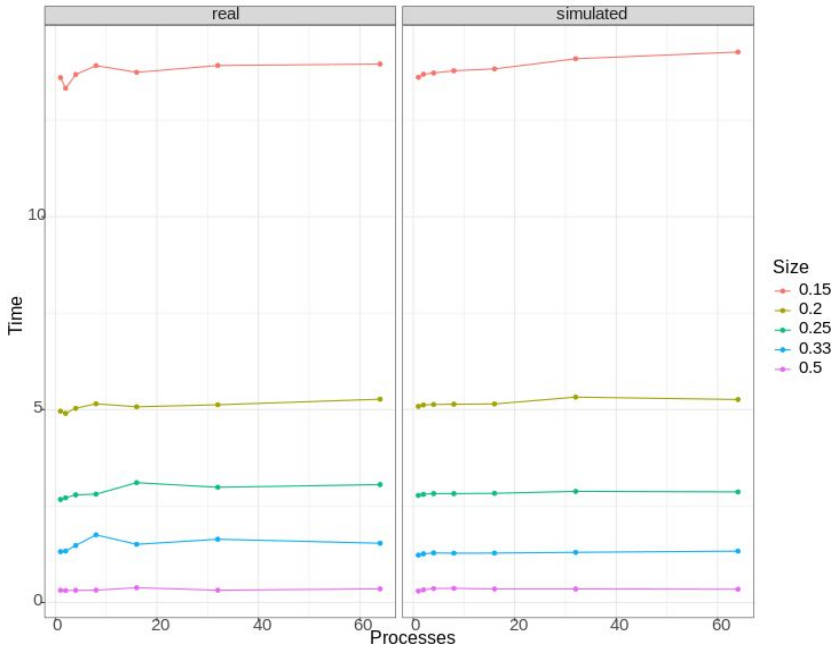


**Figure 11:** Comparison of reported total execution timings (seconds) for a libconv-based representative BigDFT operation in "default" mode (real), and its simulated SMPI estimations in "injection" mode (simulated). 150 libconv kernels are executed or simulated for each run. Each color represents a different computation size and runs were performed on 1 to 64 MPI processes (16 nodes) with 8 OpenMP threads per MPI process (weak scaling, this operation does not involve intensive communication operations). Simulation was performed on a single node, by having libconv inject computing times for each internal kernel. These times were computed at runtime, based on the coefficient file generated from a previous single process "benchmarking" run (no noise was injected in these runs).

This allows for fast and accurate simulation of libconv kernels, as shown on Fig. 11. Combined with memory folding, this reduces simulation cost a lot, making it possible to run large simulations quickly on a single node, with various settings, to get accurate advice before running costly computation on multiple nodes.

The behavior of this resulting "libsimconv" version of libconv library is being studied currently, but shows promising results already. This work has been presented at SIAMPP20 in Seattle in February 2020 as part of the "The Many Faces of Simulation for HPC" symposium.

## 3.5 CP2K

In the past months the work has been focused on integrating COSMA library and its **pdgemm** wrapper into CP2K code and verifying the performance of the new

implementation in the RPA calculations of a 128 water molecule system. We performed the runs on 128 and 1024 nodes of Piz Daint and collected the data listed below.

| **pdgemm problem size** | | |
|---|---|---|
| **M** | **N** | **K** |
| 17408 | 17408 | 3473408 |
| **Block dimensions** | | |
| **BLOCK M** | **BLOCK N** | **BLOCK K** |
| 8704 | 8704 | 13568 |

**Table 6:** Dimensions of the matrix-matrix multiplication problem arising in the RPA calculations of a 128 water molecule system.

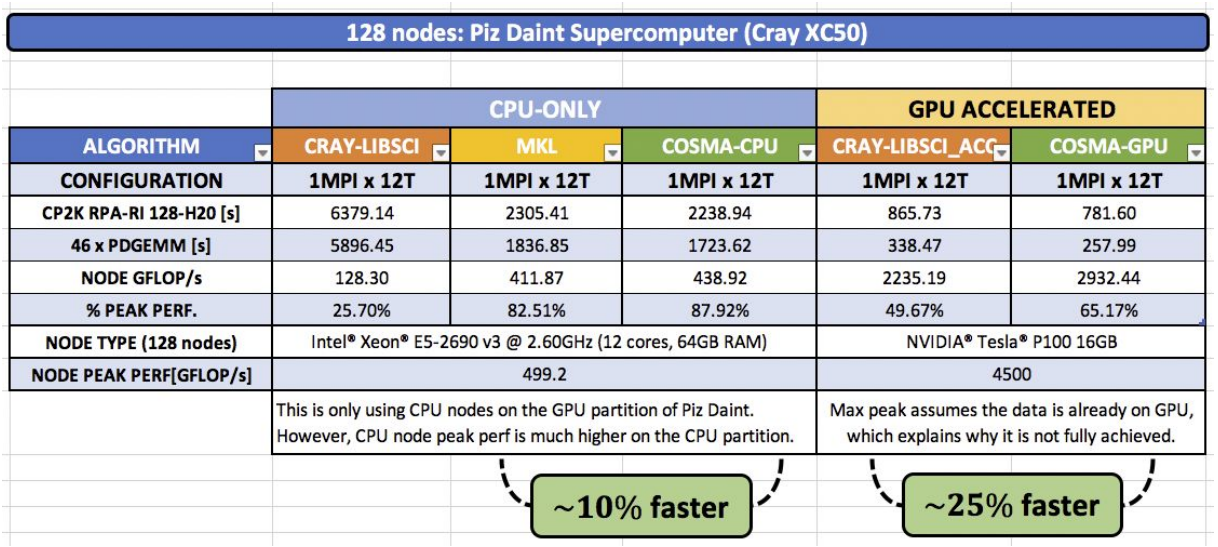


**Figure 12**: Performance on 128 nodes of Piz Daint. COSMA library outperforms MKL on CPU nodes and Cray's accelerated LibSci_acc on the GPU nodes. We were able to achieve 65% of peak performance on the hybrid GPU nodes.

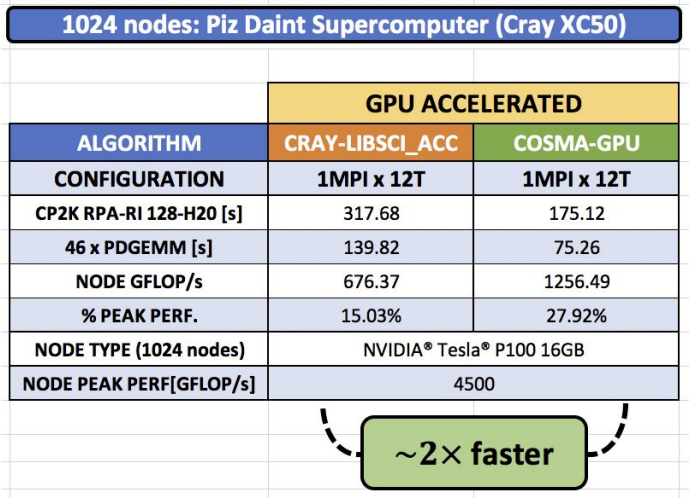


**Figure 13**: Performance on 1024 nodes of Piz Daint. COSMA library outperforms Cray's accelerated LibSci_acc library in **pdgemm** calls by a factor of ~2.

## 3.6 SIESTA

As discussed in the previous deliverable D4.2, the main opportunities for further optimization in SIESTA revolve around the solvers, which obtain the density-matrix (representing the electronic structure) from given Hamiltonian and overlap matrices.

The main breakthrough since then has been the addition of GPU support for diagonalization through the use of the GPU-enabled version of the ELPA library.

The strategy to use GPU-enabled solver libraries for GPU acceleration in SIESTA is an obvious one, since the solving step usually takes the lion's share of the execution time. Also, the non-solver part makes heavy use of indirection for the handling of sparse matrices, so it is not very amenable to GPU acceleration.

The GPU acceleration feature is available in already released versions of the code (since 4.1), using a direct interface to ELPA. ELPA has had GPU support for a while (for the 'one-stage' flavour of the solver[3]) and has been recently enhanced to add GPU support to the 'two-stage' flavour[4]. This extension effort is being done in collaboration

[3] P. Kuˇs, A. Marek, S. Koecher, H.-H. Kowalski, C. Carbogno, C. Scheurer, K. Reuter, M. Scheffler, and H. Lederer, "Optimizations of the eigen- solvers in the elpa library," Parallel Comput. 85, 167 – 177 (2019)

[4] Victor Wen-zhe Yu, Jonathan Moussa, Pavel Kůs, Andreas Marek, Peter Messmer, Mina Yoon, Hermann Lederer, Volker Blum, "GPU-Acceleration of the ELPA2 Distributed Eigensolver for Dense Symmetric and Hermitian Eigenproblems,", arXiv:2002.10991 (https://arxiv.org/abs/2002.10991)

with the ELSI project. Since the MAX-1 M12 release, which featured the new ELSI interface, Siesta is also able to run on GPUs through the ELPA solver in ELSI.

We will see below that benchmarks carried out for a range of systems on the new Marconi 100 system at CINECA show sizable speedups. For all systems, the cost of the non-solver part of the execution (the setup of the Hamiltonian) is very small compared to the solver part (less than 5% initially). Hence, speedups in the solver are basically speedups in the overall calculation. This is a very significant development, and a milestone for SIESTA.

Marconi100 has nodes composed of two 16-core Power9 processors, and four Volta GPUs. In the benchmarks reported in this section, we have used the Spectrum MPI library, CUDA version 10.1, and IBM's optimized ESSL library, with the GNU 8.4 compilers. Our resource unit below is the node, which corresponds to 32 MPI tasks (and optionally 4 GPUs). We do not take advantage of the 4 extra hyperthreads per core offered by the Power architecture.

We use two versions of SIESTA for the benchmark. The first is **4.1-b4-133**, which includes an interface to the ELPA library, including options to exercise the GPUs, but with some limitations to preserve the general diagonalization data structures in the code (the ELPA solver is used not only in the scf cycle, but in many other parts of the code as well). With this version we use ELPA 2020.05.001-rc1. The second SIESTA version is **MAX-1.0-14**, produced as part of the M12 deliverable of the project. It includes an interface to the ELSI library, which in turn offers GPU-accelerated ELPA as one of its solver options. ELSI offers an integrated workflow for the solution of the Kohn-Sham problem, and its internal data structures can take fuller advantage of the capabilities of the ELPA solver. The version of ELSI used has the date stamp 20200429, and it is one of the first with GPU support in its integrated version of ELPA (so no external ELPA library is needed).

### 3.6.1 A first CPU-GPU benchmark and analysis

The first benchmark uses a system composed of several images of a large Si quantum dot saturated with H (CSIRO benchmark Si987H372[5]). The base system contains 1359 atoms, and for the purposes of the benchmark we replicate it 8 times. With a minimal basis, this results in a problem with around 35000 orbitals.

We compare in Fig. 14 the performance of the CPU and GPU versions. Speedups range from 2.7x to 1.9x for the SIESTA-ELPA (4.1) version and from 4.0x to 2.3x for the ELSI

[5] Barnard, Amanda; Wilson, Hugh (2015): Silicon Quantum Dot Data Set. v2. CSIRO. Data Collection. https://doi.org/10.4225/08/5721BB609EDB0

(MAX) version. Speedups are typically lower at larger node counts, since the GPUs are progressively further from full saturation.

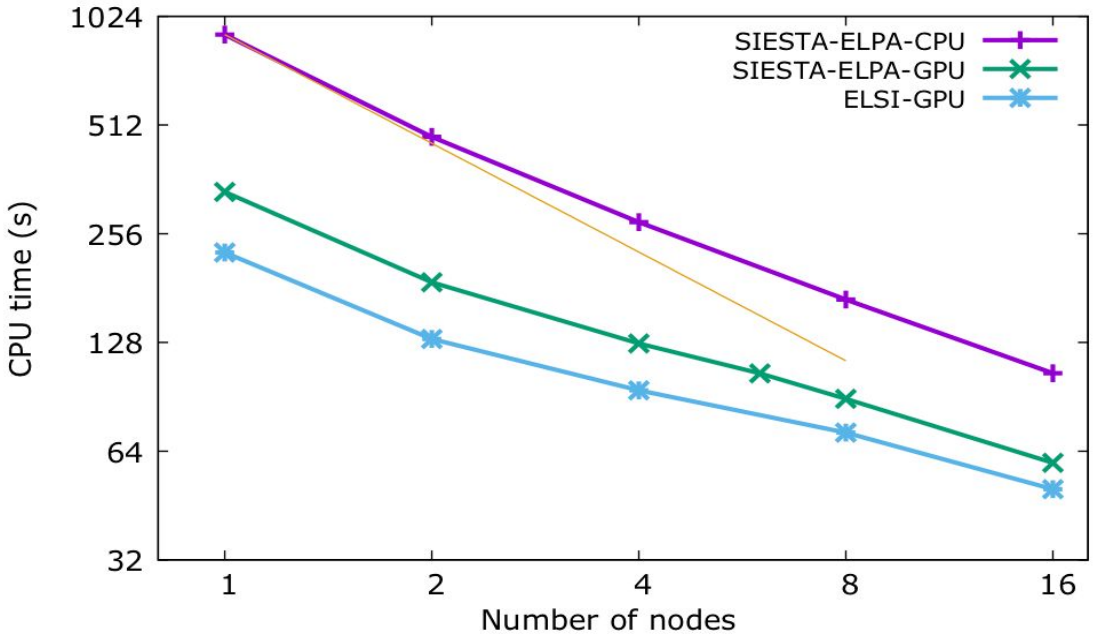


**Figure 14:** Time to solution for the diagonalization problem based on a large Si quantum dot, with approximately 35000 orbitals. CPU means here 32 MPI tasks per node on Marconi100 (Power9 architecture). GPU values are for 32 MPI tasks per node, plus 4 Volta GPU devices per node. ELSI-GPU refers to the use of the ELPA library through the ELSI interface. The thin line shows the ideal scaling with the number of nodes. Note the double logarithmic scale.

It is interesting to decompose the overall timings to see the contribution of the individual phases of the computation. We show the results for the ELSI-ELPA case in Fig. 15. The Cholesky step factorizes the overall matrix, a prerequisite for the transformation of the generalized eigenvalue problem into a standard one (which is the second step). The main phase is the solving of this standard problem (which itself is split in several steps, as explained more extensively in the ELPA references). The original eigenvalue problem is formally completed after the back-transformation of the eigenvectors, but the full solution of the electronic-structure problem still needs the building of the density matrix (DM). We note that the Cholesky and DM-building steps have not yet been enabled for GPU acceleration, but those steps that have been ported show very significant speedups.

| | 1 node | 2 nodes |
|---|---|---|
| forward transformation | 7.1 | 6.1 |
| solution of standard eig prob | 6.7 | 5.9 |
| back transformation | 14.5 | 11.5 |
| Overall speedup 1st scf step | 4.0 | 3.7 |
| Overall speedup later steps | 5.0 | 4.6 |

**Table 7:** 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.

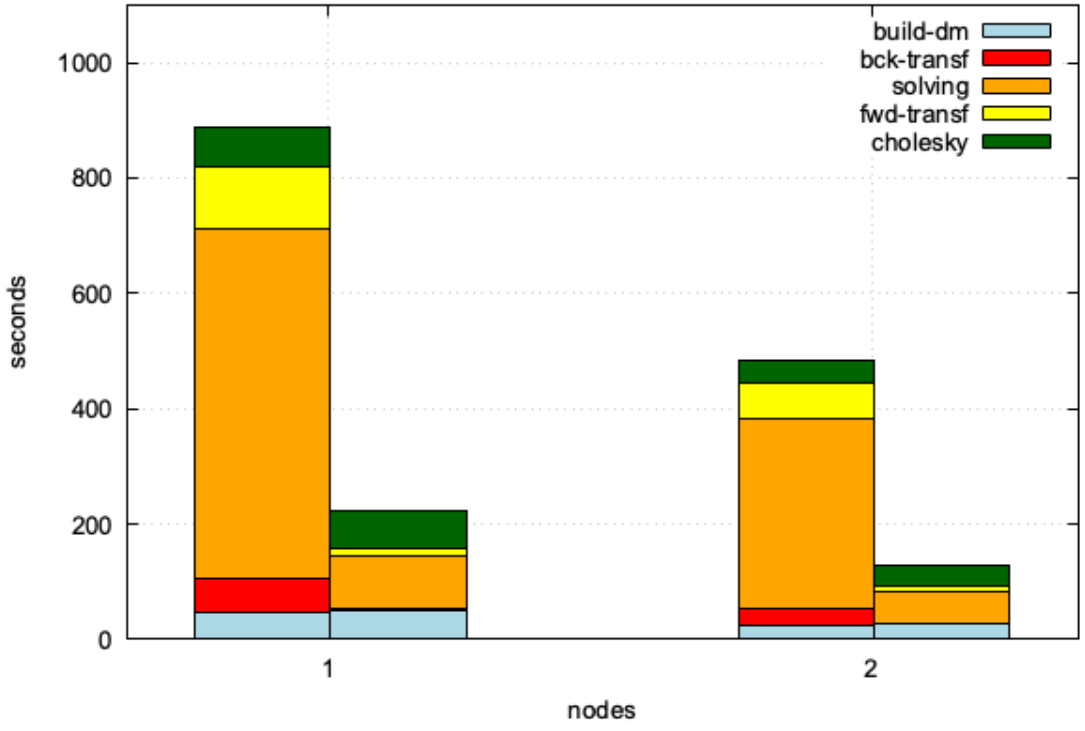


**Figure 15:** Analysis of the GPU speedup of different stages of the ELSI-ELPA solver.

These timings are based on a single scf step to conserve resources. In a real calculation, the effective speedups can be higher, since the Cholesky step does not need to be repeated in further steps if the factorization of the overlap matrix is kept

in memory. If we remove this step from the timings, the overall speedup for subsequent scf steps in one node is now 5.0x, compared to 4.0x for the first step (see also Table 7).

A similar analysis can be carried out for the SIESTA-ELPA GPU acceleration. In this case the speedups are lower because the forward and backward transformations do not take full advantage of the ELPA functionality. The acceleration of the solving of the standard problem is basically the same as in the ELSI version. This can be considered as a bottleneck that should be addressed, as currently this ELPA interface is used (even in the MAX version) for other diagonalizations outside the scf cycle.

### 3.6.2 Hermitian diagonalization

Bulk systems with k-points, and systems with non-collinear spin, need to solve a hermitian eigenvalue problem. In this case the arithmetic load, and the memory requirements, are higher. We have carried out a benchmark of a simple bulk Si system with 2048 atoms, for an off-center k-point, to evaluate the relative performance of the CPU and GPU versions.

We can see in Fig 16 that the basic behaviour already seen in the real symmetric case is maintained: the speedup obtained with GPU acceleration ranges from 3.5x for 1 node to 2.1x for 8 nodes. Again, if we remove the Cholesky decomposition phase from the accounting, the speedups for scf steps beyond the first are increased to 5.5x (1 node) and 2.5x (8 nodes).

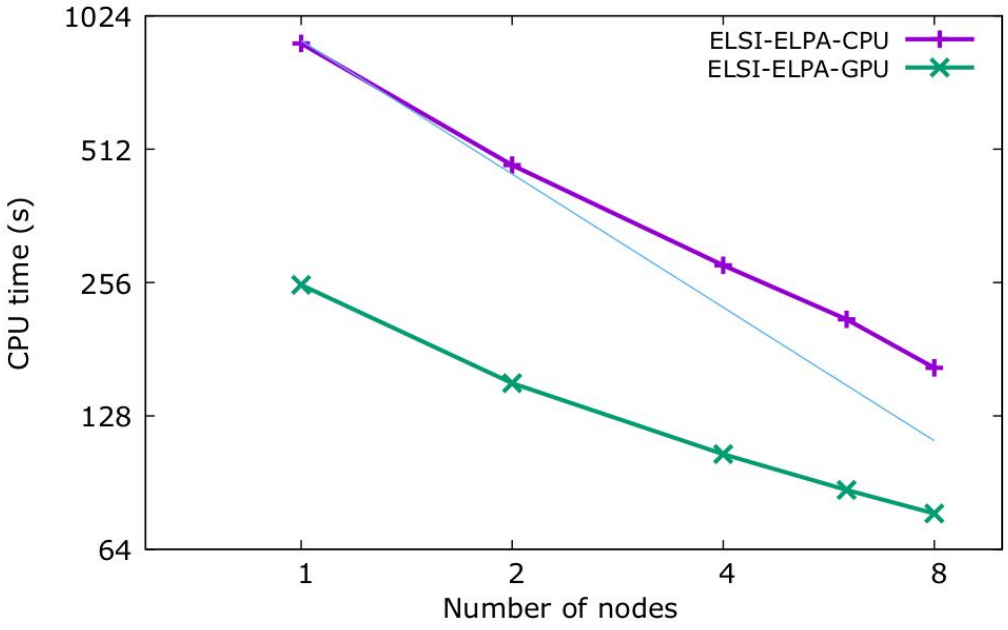


**Figure 16:** Time to solution for a hermitian diagonalization problem, for bulk Si with 2048 atoms and approximately 27000 orbitals. CPU and GPU usage details as in Fig. 12.

### 3.6.3 A very large system: sars-cov-2 protein in water

To check the new accelerations in an even larger system, we have run benchmarks on the same Sars-CoV-2 envelope protein mentioned in the Quantum ESPRESSO section. The details of the structure, box size, and real-space-mesh cutoff (density fft for QE) are the same. Obviously, the basis set in SIESTA is different. We have run a simple example with a minimal basis (22300 orbitals) just for basic checks, but the bulk of the benchmarks have been done including polarization orbitals, for a total of approximately 58000 orbitals. This is almost double the size as the previous benchmark.

One first issue to consider is that the memory requirements are such (they scale as the square of the matrix size) that the problem does not fit in a single node of Marconi 100. Hence the data on Fig 17 starts at two nodes. We have used the MAX-1.0-14 version of SIESTA with the ELSI library. The GPU speed-up for 4 nodes is 3.4x. In the figure we have also included a line for the PEXSI solver, which will be discussed in its own section below.

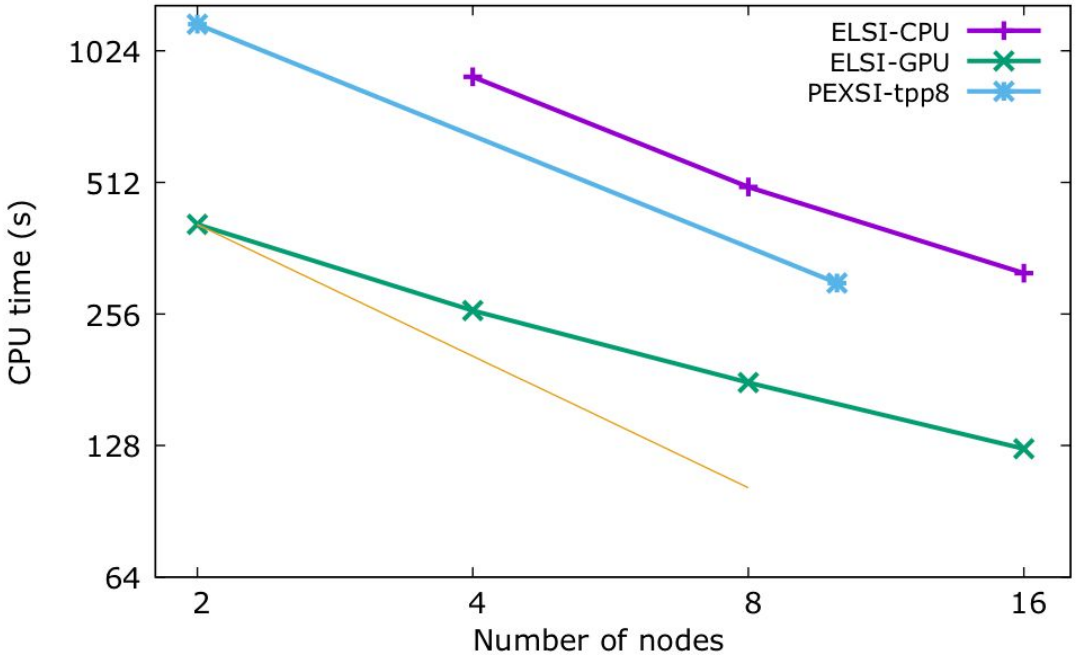


**Figure 17:** Time to solve the diagonalization problem corresponding to a piece of sars-cov-2 protein surrounded by water molecules, with approximately 58000 orbitals. CPU and GPU usage details as in Fig. 12. For the meaning of the PEXSI line, refer to the discussion below.

### 3.6.4 Relative performance of GPU-accelerated diagonalization and PEXSI solver

The SIESTA solver of choice for massively parallel calculations for large systems has been PEXSI (Pole EXpansion and Selected Inversion), due to its favorable size scaling, multi-level parallelization scheme, and smaller memory needs. SIESTA was the first mainstream code to offer an interface to the PEXSI library, and now further enhancements are available through the ELSI layer.

With the availability of GPU-enabled versions of SIESTA, it is relevant to revisit the issue of the placement of the "break-even" point. Fig 16 seems to show that the GPU-accelerated diagonalization solver is much more efficient than the PEXSI solver. However, this has to be qualified, and placed in the fuller context of the user needs. If minimization of the time-to-solution is the main goal, then the more favorable scaling of the PEXSI solver is key.

We will use the protein system with SZP basis as our benchmark. In Fig. 16 it was apparent that the scaling of the GPU-accelerated solver was rather degraded already at 16 nodes (which, by the way, was our node limit in the benchmarks to conserve resources). The PEXSI line was tagged "tpp8", which means that 8 MPI tasks per pole were used. For the PEXSI calculation in Fig 14 we used 20 poles for the expansion of the Fermi-Dirac function. Taking into account that the PEXSI solver parallelizes also over chemical potential points (two in this case), with two nodes (64 cores) we can process 4 poles simultaneously. Five sequential batches of 4 poles cover the total calculation. With 10 nodes we can process all 20 poles at the same time. Hence the PEXSI line corresponds to a nearly-trivial parallelization, and shows nearly ideal scaling (not completely, as discussed below).

Beyond parallelization over poles, we have more scaling scope in the tpp parameter. For each pole and chemical potential, a number of MPI processes are assigned to carry out the selected inversion algorithm. There is in practice a lower limit for tpp: since each team of processes needs to have full copies of the Hamiltonian and overlap matrices, tpp cannot be too small, or else the node memory would be exhausted. In this benchmark, tpp=8 is the minimum value. Four copies of H and S (and auxiliary data) are kept in each node. These are sparse matrices, and their size is (sparsity*N*N), where the sparsity is approximately 0.012, and N=58000. There is in principle no upper limit for tpp, and using progressively larger values is the route for massive parallelization.

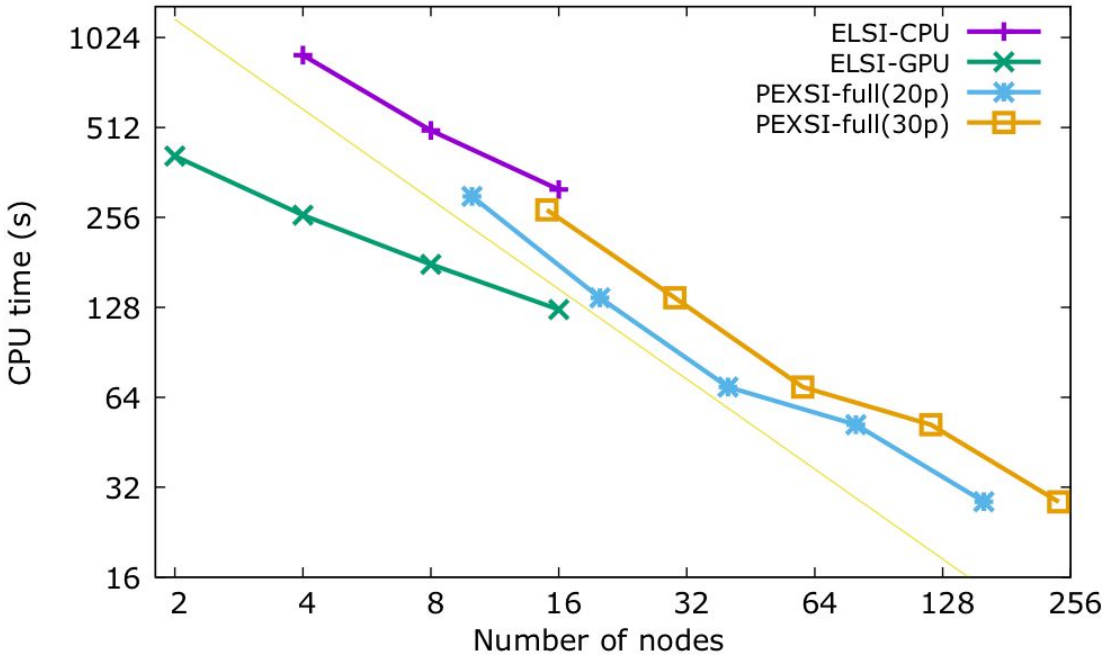


**Fig. 18**: Time to solve the diagonalization problem corresponding to a piece of sars-cov-2 protein surrounded by water molecules, with approximately 58000 orbitals. CPU and GPU usage details as in Fig. 12 . Two sets of PEXSI results (for 20 and 30 poles) are shown. The thin line shows the ideal scalability behavior.

In Fig. 18 we advance the final result: the very good scaling reserve of the PEXSI method means that it can use effectively many more tasks to provide much lower times-to-solution than the GPU-accelerated diagonalizer. We still need to explain how we obtained the data for the PEXSI-full lines (here “full” is meant to refer to using the full parallelization possibilities of the solver), since we could not use more than 16 nodes.

The key is again the near-trivial parallelization over poles. We can perform calculations with the available nodes, fitting as many poles as possible for the appropriate number of tasks per pole, and processing sequentially in batches until the total number of poles is taken care of. In Table 8 we give the details for the case in which the solver uses 20 poles in the expansion of the Fermi-Dirac function. This number of poles gives a very close approximation to the results of diagonalization, but for a full match 30 poles might be needed. The calculations for 30 poles are very similar: an extra calculation with 15 nodes for tpp=8 is needed, since the number of poles per batch for 10 nodes is not a divisor of 30. Using 30 poles instead of 20 increases the number of nodes needed to reach a given time to solution, as shown in Fig. 18.

| Tasks per pole | # of nodes in base run | # of poles per batch | cpu-time (s) of base run | # of nodes full parallel | est. time (s) full parallel |
|---|---|---|---|---|---|
| 8 | 10 | 20 | 310 | 10 | 310 |
| 16 | 10 | 10 | 275 | 20 | 137.5 |
| 32 | 10 | 5 | 277 | 40 | 69.25 |
| 64 | 16 | 4 | 259 | 80 | 51.8 |
| 128 | 16 | 2 | 286 | 160 | 28.6 |

**Table 8:** Details of the estimation of the PEXSI solver performance using 20 poles, for a problem with approximately 35000 orbitals.

While very relevant for many projects, minimum time-to-solution is not the only possible goal. Users might want to maximize the return of their supercomputer allocation by carrying out as many jobs as possible, without regard (within limits) to the time involved. In this case, minimizing the total cost (in node*hours) of a calculation is the relevant objective.

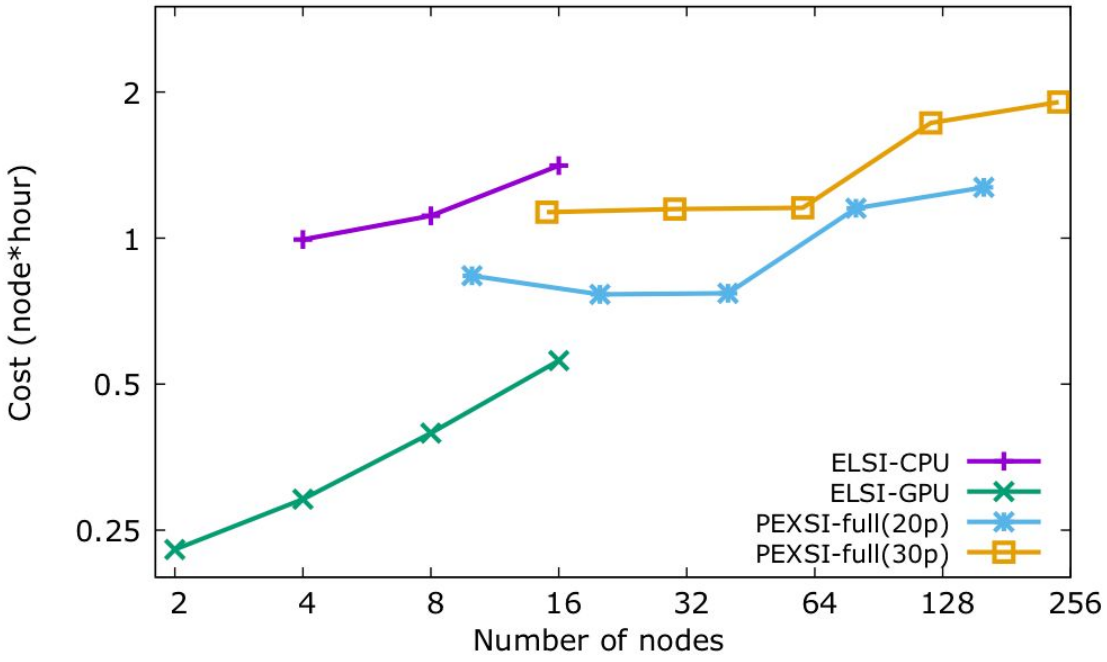


**Fig. 19**: Total cost (per scf step) for the virus protein problem, with approximately 58000 orbitals. CPU and GPU usage details as in Fig. 12 . The PEXSI lines correspond to different numbers of tasks per pole (from left to right: 8, 16, 32, 64, 128). A horizontal line in this plot would mean perfect scaling with node count.

Fig. 19 presents another view of the benchmark emphasizing cost, and also nicely showing the level of scaling of the different methods by the deviation from horizontal lines. If cost is the main concern, then the GPU-accelerated diagonalization wins (although it might be argued that nodes with GPUs should be charged at a higher rate than CPU-only nodes; this point is moot on Marconi 100, but could be relevant elsewhere).

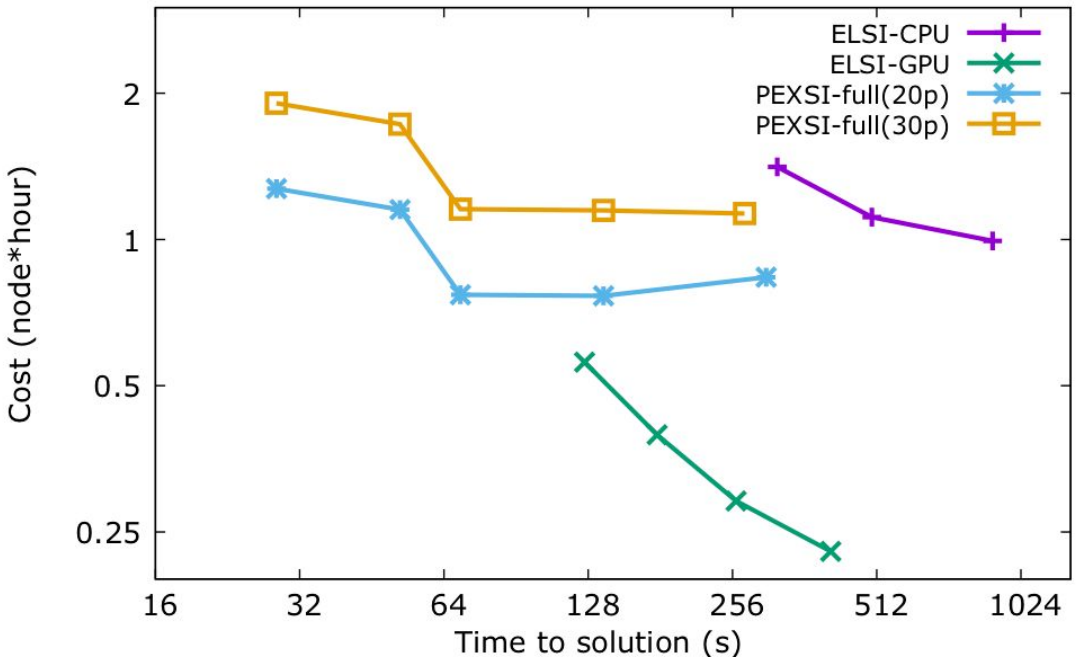


**Figure 20:** Total cost (per scf step) vs time-to-solution for the virus protein problem, with approximately 58000 orbitals. CPU and GPU usage details as in Fig. 12 . The PEXSI lines correspond to different numbers of tasks per pole (from right to left: 8, 16, 32, 64, 128).

Yet another way to look at the issues involved is provided by Fig. 20 . Here proximity to the lower-left corner represents the overall "goodness" of the method. Also, the (negative) slope of a line reflects the marginal cost of diminishing the time-to-solution, which is lower in the PEXSI method, but note that there is a sharp drop in efficiency when going from tpp=32 to tpp=64. This obviously reflects the fact that the intra-pole parallelization now needs to perform communications with other nodes, with higher latency. In this benchmark we did not go beyond tpp=128, but it would be interesting to try larger systems and see if they can maintain a good scaling

at those levels (the original benchmarks of the SelInv algorithm show that they should).

In the discussion of the first benchmark we noted that the diagonalization solver is more efficient in scf steps beyond the first, since it does not need to factor the overlap matrix again. The PEXSI solver also has some tasks that are typically done only in the first step: a symbolic factorization of the matrices involved, and an "inertia-counting" phase to provide tight bounds for the chemical potential. In this benchmark the cost of the Cholesky step in diagonalization and of the factorization and bracketing phases in the PEXSI solver are rather similar, so in order to simplify the discussion we have not carried out an extra comparison of time-to-solution and cost for scf steps beyond the first. The presence of these extra preliminary calculations is responsible for the not-completely ideal scaling with the number of nodes for a given tpp value.

A very important point is that the performance of the PEXSI solver depends on the sparsity of the system. For a relatively dense 3D system like the protein in water the sparsity is moderate. In more sparse systems the PEXSI method could offer a smaller time-to-solution than ELPA-GPU even for relatively small node counts (this has been seen in the (artificial) replicated quantum-dot of the first benchmark, although it is not reported fully here).

Moreover, the developers of the PEXSI library (private communication) are working towards enabling GPU acceleration. This would be a very exciting development, and we will be monitoring it.

### 3.6.5 A new kind of bottleneck: method and parameter choice

To close this section on SIESTA benchmarking, we would like to call attention to an emerging problem, which we might also call a bottleneck: it is not trivial to choose the right method (and parameters of the method) to maximise the efficiency of a given calculation. The problem is made worse by accelerated architectures, multiple hierarchies (core, socket, node, etc) and the concomitant need to find the right affinities. It is not enough to provide a well-optimized code. Users need to be given automated advice (in the form of heuristics, or machined-learned data) to make good use of the very complex machines that are becoming available.

## 4 AiiDA as a tool for benchmarking

One proof-of-concept (PoC) consisted in the adoption of the AiiDA framework in order to automate and standardize the benchmarking of the flagship codes. This was preliminarily accomplished using Quantum ESPRESSO for a test.

Once the test cases were defined we needed to make three different operations:

1) setup a python script to be used under AiiDA to submit the jobs;
2) setup the computer, i.e. the machines where to run the codes and to use the related plugins in AiiDA;
3) create a script to aggregate the simulations that were belonging to a same group and then use this script to perform the performance analysis;

This process highlighted some difficulties. In particular, the most difficult part has been the one described in the third point above, because in some cases AiiDA has not implemented the tools in order to parse the timings of the applications (Quantum ESPRESSO in particular) and to elaborate them. This can be however solved with the help of some scripting in python. The other difficulty, which prevented us from performing benchmarks with AiiDA in time for the submission of this deliverable, is that a non negligible amount of work is required to set up a new machine. In particular, a lot of details of the submission script need to be "hard-coded" in the submission script, losing flexibility in the usage of the tool. These aspects are better managed in tools such as JUBE, which are designed targeting on purpose for the benchmarking only. On the other side, one of the advantages of AiiDA (which is only in part present in JUBE) is the storing in the database of all executed runs, which permits to retrieve the data also executed in previous runs, maybe a long time before.

The prototype for this PoC has been publicly released and versioned in Gitlab[6].

As a conclusion of the PoC with AiiDA we can conclude that some effort is still needed to make it possible to use it as a benchmarking tool, and that this would require, in any case, a strong support from both the code owners and the staff from the computing centers.

[6] https://gitlab.com/fabioaffinito/aiida-scripts

# 5 Conclusions

This deliverable reports, among other, data from a benchmarking campaign held in spring 2020. For the first time we had the opportunity to run MaX codes on a large-scale GPU-accelerated machine, such as Marconi100 hosted at CINECA. This allowed us to benchmark the GPU-porting of MaX codes on test cases of different size (including extremely large partitions thanks to a dedicated pre-production access granted on the Marconi100 machine). This is particularly relevant in view of the expected architectures of the EuroHPC pre-eascale machines to be deployed in early 2021.

Overall, the performance of the MaX codes has been found to be already very good in most cases, demonstrating the capability to run profitably even on very large GPU-accelerated partitions. Moreover, the benchmark data reported in this deliverable open the way to further improvements in the performance (and performance portability) of the codes. In particular, two important directions for the work on the codes have been identified :

1) a challenge posed by GPU architectures: besides the aspects related to the programming models, given the large compute-power provided by the GPUs, and the good performance of the codes on the most time-consuming kernels, unforeseen memory and time bottlenecks are emerging in parts of the codes that, up to this moment, were not a problem on classical CPUs architectures; These need to be explicitly addressed to make the performance portability more uniform across the codes.
2) the adoption of performing libraries (GPU-aware distributed linear algebra, COSMA, Libconv) is becoming crucial to operate in complex environments taking into account the concept of “separation of concern”.

Both these aspects are transversally present in all the MAX flagship codes and will be the main target of the work in the next months.