D2.1

First release of MAX software: report on the performance portability

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

Due date of deliverable	30/11/2019 (**month 12**)
Actual submission date	29/11/2019
Lead beneficiary	JUELICH (participant number 4)
Dissemination level	PU - Public

Document information

Project acronym	MAX
Project full title	Materials Design at the Exascale
Research Action Project type	European Centre of Excellence in materials modelling, simulations and design
EC Grant agreement no.	824143
Project starting/end date	01/12/2018 (month 1) / 30/11/2021 (month 36)
Website	http://www.max-centre.eu
Deliverable no.	D2.1
Authors	Daniel Wortmann, Uliana Alekseeva, Stefano Baroni, Augustin Degomme, Pietro Delugas, Stefano de Gironcoli, Andrea Ferretti, Alberto Garcia, Luigi Genovese, Paolo Giannozzi, Anton Kozhevnikov, Ivan Marri, and Nicola Spallanzani.
To be cited as	Wortmann et al. (2019): First release of MAX software: report on the performance portability. Deliverable D2.1 of the H2020 CoE MAX (final version as of 29/11/2019). EC grant agreement no: 824143, JUELICH, Germany.

Disclaimer

Contents

1 Executive Summary

This report summarises the efforts and the progress concerning performance portability, made during the first 12 months of the MAX Centre of Excellence (phase 2). The results are included in the corresponding releases of the flagship codes. According to the general goals of WP2 and the MAX Software Development Plan, we focused on enhancing the support for future and emerging computing architectures and on the development of strategies to make performance available for codes and users.

A major focus in this first phase of the work was on the exploration of the challenges imposed by different computing architectures and the performance that was already achieved, as well as the identification of code in need and suitable for implementation of performance portable approaches. We basically followed two main thrusts: on one hand, we investigated the MAX flagship codes and their most computationally relevant parts, on the other hand we augmented this by a bottom-up study based on independent libraries and relevant computational kernels, including those developed in WP1. Concerning computational architectures, a major focus was on GPU-based accelerators and the corresponding programming frameworks like CUDA or ROCm. Additionally, we started to investigate particular aspects related to the ARM architecture. Besides code sections relevant for standard DFT tasks, our analysis also included computational workloads that are relevant for beyond-DFT methods (such as GW), as implemented in YAMBO, or included in terms of hybrid functionals in other flagship codes. As many of the kernels involved in these approaches are computationally very demanding, several of them have already been optimised for different computing architectures: we will continue to work on their overall performance portability.

In summary, we implemented, investigated and demonstrated the efficient use of a large diversity of current computing architectures and software development frameworks for our codes and libraries as developed out of WP1. The most significant parts of our codes already run on various of such platforms, including heterogeneous architectures, and some parts already demonstrate satisfactory performance. Noticeably, a fully-fledged GPU-aware implementation of GW is made available by the YAMBO v4.5 release. In the future, the focus will shift to the unification of different platform-specific implementations, to the exploitation of the further performance optimisation opportunities that we identified, and to the deployment of performance portable libraries and codes.

A second activity within WP2 is the development of auto-tuning tools, of software to use different hardware efficiently, and of tools to monitor the performance of the codes and to detect shortcomings. In this context, we have developed performance models for some of the MAX flagship codes (QE, FLEUR, BigDFT). While these models are of course of fundamental importance in the context of high throughput computing in WP5, they also provide many valuable information on performance bottlenecks on particular architectures, an optimal use of resources and some parallelisation strategies up to the point in which the code can auto-tune such parameters. For example, the assignment of available computational resources to different sub-tasks within a simulation run is generally a complex problem depending on the physical details of the materials system under investigation, on the properties to be evaluated, on the total resources available and on their computational capabilities and hardware specifics. Our performance models are thus an indispensable component to enable the efficient use of the performance-portable kernels and solutions we develop.

2 Introduction

WP2 aims at the delivery of codes which offer optimised performance for existing and emerging architectures with various levels of heterogeneity. Moreover, the codes should deliver optimal performance regardless to which goal additional optimisation is adapted to– be it energy-to-solution or time-to-solution. Moreover, all this optimisation richness should be easily available to the end users, ranging from developers and HPC experts, to academic and industrial application users.

The task of delivering performance optimised code requires both a detailed knowledge of the involved codes and algorithms and a critical review of the status of the implementation and the adaptation of new programming paradigms. Such performance optimisation, once designed for a single architecture, is conceived now for many different platforms, including currently available hardware, and able to precede the needs of future concepts. Hence, to accomplish such ambitious goal, many steps out of which the establishment of a firm code base with demonstration of high performance on various architectures was a first focus.

The increasing diversity of computing platforms supported by our codes not only imposes issues to the developers and motivates our quest for performance portable solutions, but also challenges the users as it becomes increasingly difficult to assign the most appropriate computational resources to a given simulation task. To overcome this obstacle we started to build performance models dedicated to our codes to enable the user in making well-informed choices regarding the setup of their simulations. We foresee that this activity will develop into more automatic tools able to perform auto-tuning tasks.

This deliverable reports about the state-of-the-art of flagship codes in this context and is divided in three sections. First, the outcome from the exploration of various computer architectures, massively including heterogeneous systems, is presented. Then, examples of performance portability achieved by several libraries are given. Finally, first results on performance prediction and auto-tuning are presented.

3 Exploration of computing architectures and code specific performance portability

First we report on the performance our flagship codes achieve on different computing platforms and the corresponding implementations.

3.1 Quantum ESPRESSO

The QUANTUM ESPRESSO distribution may be compiled and run efficiently on all systems based on hybrid MPI + openMP parallelism. In particular it has been extensively tested on systems based on Intel, ARM, and IBM Power processors (see fig. 1 and table 1). Since release 6.4.1, a fully functional version of `pw.x` – the main quantum engine of QUANTUM ESPRESSO – can also be compiled and run on systems based on hybrib MPI + GPU acceleration on architectures based on the CUDA GPUs. The porting to such platforms has been done using the CUDA-FORTRAN programming model.

The acceleration of the two most compute-intensive kernels of `pw.x`, namely FFTXlib for 3D FFTs and LAXlib for parallel linear algebra, has been found to be crucial for the

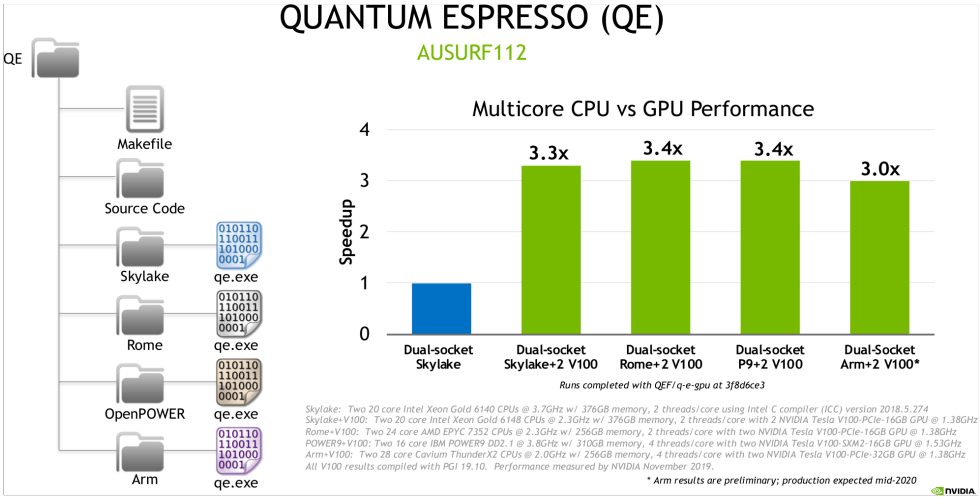

Figure 1: Speedup of QUANTUM ESPRESSO on different architectures equipped with NVIDIA Tesla V100 cards. Courtesy of Josh Romero and Massimiliano Fatica from NVIDIA.

performance portability. Concerning the technical aspects of the porting, the LAXlib library on GPUs exploits the GPU-aware libraries provided by NVIDIA and PGI. FFTXlib performs the acceleration at various levels depending on the size of the FFT mesh. If the whole mesh fits into the memory of a single GPU, then a 3D FFT is performed directly by the CUDA specific kernel for 3D FFT. If instead the mesh has to be distributed on multiple MPI tasks, the GPU acceleration is obtained by performing local 1D and 2D FFT operations and data is then scattered *via* MPI to complete the 3D FFT operation. In order to reduce latency times, FFTs on wave functions are performed in batches, allowing the code to overlap the MPI communications on one batch with the GPU FFT operations on another batch.

The performance of `pw.x` on different machines for a selected benchmark case is presented in table 1. Present CINECA HPC Cluster Marconi is compared with different CPU+GPU setups. The comparison demonstrates a good performance portability of `pw.x` towards hybrid systems based on accelerators. See also [1]. As already discussed in the D4.2 deliverable [1], the *pool* parallelism on **k**-points is completely portable and the FFT parallelism performs quite satisfactorily. The most important concern for portability is related to the parallel linear algebra used during the iterative Davidson diagonalisation. The linear space on which linear algebra is performed in this case (the iterative space) may become too large and parallel linear algebra can not be performed using a single GPU device. Answers to this concern are expected from two sides. On the algorithmic side we are working [2] at the development of alternative diagonalisation algorithms that avoid an excessive increase of the iterative space. On the kernel side, the performance portability will benefit by any specific kernel able to perform parallel linear algebra distributing the matrices on many GPU devices.

[1] https://gitlab.hpc.cineca.it/PPI4HPC/benchmark/tree/master/QuantumESPRESSO

	Marconi	Galileo	Bench system 1	Bench system 2	Bench system 3
nodes	64	12	16	16	4
Proc model	Xeon PHI	Xeon Haswell	Xeon Skylake	IBM Power9	Xeon Cascadelake
# cores	4096	192	320	256	64
GPU model	-	K80	V100	V100	V100
# GPUs/node	0	2	4	4	4
# GPUs	0	24	64	64	16
Time [sec]	3465	5880	540	617	2473

Table 1: QE performance on different architectures with and without GPUs and with different processors and GPUs. Benchmark input is CsI with 96 atoms and 89 k-points.

3.2 FLEUR

The current (MAX -4) release of the FLEUR code already runs on different HPC platforms by using optimised basic math libraries, by a portable build process with system-specific build options and by specific implementations of computational relevant kernels tuned for different architectures. In particular, the code sections discussed in detail in WP1 and WP3 - like the matrix setup or the interface to the linear algebra - already take into account the requirements of different hardware concepts and show reasonable performance and scalability. To identify additional performance portability problems and opportunities, we investigated the runtime of FLEUR on various machines and architectures. We put the focus of our study on two aspects.

In the first line of study, we compared several machines with very similar computing architecture, all based on standard Intel Haswell server processors. In detail, we performed timing runs on CLAIX 2016, a machine at the RWTH-Aachen University, on the JURECA cluster module at the Juelich Supercomputing Center, on Hazel Hen, a machine at the High Performance Computing Center Stuttgart and on SuperMUC at the Leibniz-Computer Center in Munich. On each of these systems we performed simulations using different test setups developed in collaboration with the effort of WP4. The timings for a single self-consistency cycle with different levels of parallelisation is reported in Tab. 2. In addition we also investigated the Intel Skylake and Intel KNL micro-architecture (Tab. 3). Here we used the CLAIX-2018 machine at RWTH, the SuperMUC-NG in Munich for two different Skylake processors and the Booster module of JURECA in Juelich for the KNL architecture.

The key findings of these simulations are:

- While the different systems show similar timings, there are also variations.
- The different systems show a slightly different scaling behaviour.
- Most of these differences can be tracked down to differences in network and IO relevant operations.
- Performance of the FLEUR code is stable and reproducible on the Intel architectures including very large setups (several thousands atoms).

Machine		CLAIX 2016	JURECA	Hazel Hen	SuperMUC
Chip		E5-2650v4	E5-2680v3	E5-2680v3	E5-2697v3
#cores/node		24	24	24	28
Frequency, GHz		2.2	2.5	2.5	2.6
Perf./node, GFlops		840	960	960	1160
use case	#nodes				
GaAs	8			1622.32	
512	16			943.45	
atoms	32	685.31	588.7	623.32	636.21
	64		403.99	446.84	
TiO2	16	3657.68			
1078	32	2172.22		1822.45	
atoms	64	1295.6	1103.03	1149.45	
	128	860.49		769.49	
	256	620.36		580.73	
TiO2	128	6235.64	5915.2	4468.9	
2156	256	3814.74		3117.04	
atoms	512	2477.9		2161.68	

Table 2: Runtime in seconds for the test cases listed on the left of different Haswell machines.

While the performance tests reported so far concentrate on machines with Intel processors using the Intel compiler tool-chain and partly very similar system design and properties, in a second line of study we investigated the performance of the FLEUR code on very different architectures. We hence performed additional investigations on a system equipped with ARM-based processors, the Hi1616 processor as used in the JUAWEI system in Juelich, and a node of the CLAIX machine with a NVIDIA GPU (Tesla P100). Again, we used the Haswell and the KNL processors as reference. The detailed setups of the different machines are listed in Table 4.

The timings of a test setup (NaCl,64 atoms, 1 k-point) on these four very different computer architectures are listed in Table 5, detailing the whole self-consistency iteration step as well as its most time-consuming constituents. All runs were exploiting only shared-memory parallelism, i.e. one process were spawned onto all cores of every chip. We found that the performance of the FLEUR code varies significantly across these architectures. The theoretical peak performance of the ARM node is roughly 70% of that of the Intel node (Table 4), but performance has dropped more than 5 times. This indicates that our current implementation does not exploit the capabilities of the ARM architecture sufficiently and additional performance tuning has to follow. Since we so far did not implement any ARM specific code, we believe that the creation or adaptation of such specialised kernels will enable increased performance in future.

The situation is somewhat different for the GPU version of the FLEUR code. Here the two most computationally intensive parts of the code, matrix setup and diagonalisation, are ported onto GPU. The timings of these parts, especially the diagonalisation, look very promising: it is 3 times faster on a GPU node than on the Intel CPU. When the peak performances are taken into account though (that of GPU being six times larger than that of Intel CPU), the necessity of further investigation and optimisation becomes clear. The extension of the GPU version to the remaining parts still poses a significant problem

Machine		CLAIX 2018	SuperMUC-NG	JURECA Booster
Chip		Xeon Plat. 8160	Xeon Plat. 8174	Xeon Phi 7250-F
#cores/node		48	48	68
Frequency, GHz		2.1	2.4	1.3
Perf./node, TFlops		3.2	3.69	2.83
use case	#nodes			
TiO2	16	2063.79		
1078	32	1279.29	1347.26	1998.47
atoms	64	801.51	855.91	
	128	670.02	617.17	
TiO2	64	6015.91		
2164	128	4126.54	3508.69	8396.63
atoms	256	2915.11	2411.18	
	512	2428.89	1858.86	
SrTiO3	128	8036.37		
3750	256	8613.69		
atoms	512	8983.76		9963.31

Table 3: Runtime in seconds for the test cases listed on the left for Skylake machines and the KNL-Booster of JURECA.

Machine $[Name]$	Chip $[Model]$	# cores	Peak Perf. $[GFlops]$
Hazel Hen	Intel E5-2680v3	24	840
CLAIX 2018 GPU	Tesla P100	1792	5300
JUAWEI	Hi1616	64	614
JURECA B.	Intel Xeon Phi 7250-F	68	2830

Table 4: Four different computer architectures which simulations with FLEUR were compared on.

to overcome in the next months as these parts contain a large variety of algorithms and hence no simple porting strategy can be applied.

3.3 Yambo

The YAMBO code has been installed and tested on a large number of HPC architectures including homogeneous multi-core systems and heterogeneous GPU-accelerated machines. From the YAMBO 4.0 version on, a deep refactoring of the parallel structure has been put in place in order to take full advantage of nodes with many-cores and a limited amount of memory per core. In particular, a MPI multi-level (up to 3–5 according to the runlevel) approach has been adopted, together with an OpenMP coarse grain implementation. Recently an intense activity of porting on heterogeneous architectures, GPUs in particular, have addressed the main kernels of the code. These include the subroutine computing dipoles, Coulomb cutoff, Hartree-Fock, linear response, GW, and Bethe-Salpeter equation (BSE).

The porting on GPUs has been performed by using the CUDA-FORTRAN programming model, which provides a native support for NVIDIA architectures. In particular

Architecture	Intel Broadwell	GPU (Pascal)	ARM	Intel KNL
Compiler	Intel 18.0	PGI 18.4	gfortran 7.3	Intel 18.2
Whole iteration, sec	36.54	123.34	192.38	50.69
Potential , sec	6.14	34.60	36.07	22.23
Matrix setup , sec	3.94	6.71	22.89	4.04
Diagonalisation, sec	18.54	6.68	118.34	14.51
Charge, sec	4.36	72.80	10.98	8.38

Table 5: FLEUR performance of a test case NaCl (64 atoms, 1 k-point, 1 self-consistency iteration step). Timings of the whole iteration as well as of its most time-consuming constituents are shown. For GPU, only matrix setup and diagonalisation are ported onto GPU.

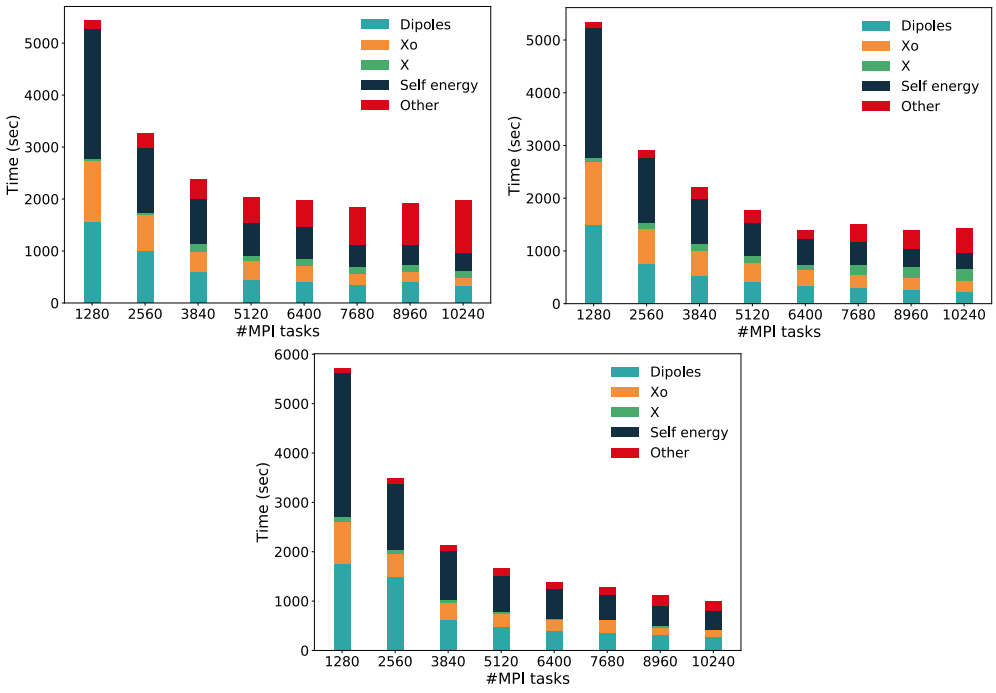

Figure 2: YAMBO: Complete GW workflow for a defected H-TiO_2 supercell (72+1 atoms). Panels show the Optimisation of the MPI usage from version 4.3 to version 4.5 (the one currently released). The first two graphs were reported in D4.2 showing the occurrence of a MPI-related bottleneck (top left panel) and a partial fix to the problem (top right panel). A complete fix is provided in the bottom panel (v4.5).

Machine [$Name$]	Chip [$Model$]	Clock [GHz]	# cores	GPUs [$model$]	Peak Perf. [$GFlops$]
Marconi-A2	Intel Xeon Phi7250 KNL	1.4	68	–	$\sim$ 3000
PizDaint	Intel Xeon E5-2690 v3	2.6	12	P100	4760
Galileo	Intel Xeon E5-2697 (BDW)	2.3	36	V100	7800
Corvina	Intel Xeon Silver 4208	2.1	16	Titan V	7450

Table 6: Different computer architectures used to benchmark YAMBO.

Architecture	**Dipoles**	χ^0	χ	Σ_x	Σ_c	**wall time**
MARCONI-KNL	163	3601	9	197	3346	8014
Piz Daint CPU (pgi)	194	10191	7	317	5221	16631
Piz Daint CPU+GPU	168	1256	2	47	168	2075
Galileo CPU (ifort)	61	5402	6	107	645	6484
Galileo CPU (pgi)	233	6874	43	378	2703	10507
Galileo CPU+GPU	163	905	11	31	118	1451
Corvina CPU (pgi)	163	7321	5	221	3937	12239
Corvina CPU+GPU	142	993	3	35	114	1639

Table 7: YAMBO: Time-to-solution on different architectures for the AGNR-N7 use case. All times are given in sec.

YAMBO makes large use of CUDA-FORTRAN cuf-kernel directives as well as of CUDA libraries such as cublas, cufft, and cusolver. Overall, the porting strategy has been based on reading and sotring DFT wavefunctions on the GPU memory (feasible since YAMBO fully distributes at the MPI level such memory), making them available for heavy computaitonal kernels. Similarly, the response function is also calculated and temporarily stored on the card memory. As a design principle, in order to improve the performance of the porting, the number of data-transfers between host and device have been minimised. Then, reduction operations, such as those needed to compute quasi-particle corrections are performed on the GPUs and the final results moved to the host memory and saved. Overall, due to the modularity of YAMBO and to the use of `devicXlib`, the use of CUDA-FORTRAN had a small impact on the code sources, the accelerated parts with replicate sources being localised only in a few routines. Noticeably, the optimisation of the code on the GPUs has also permitted to improve the performance of YAMBO on the CPUs, for instance by reducing the execution time of the `FFT_setup` and the `cutoff Coulomb potential` routines.

In Table 7, we report both the timing of the main routines (dipoles, non-interacting response function χ^0, reducible response function χ and the exchange (x) and correlation (c) part of the self-energy Σ) and the wall-time for a complete GW calculation performed for a AGNR-N7 graphene nanoribbon (see e.g. Ref. [3] for a related publication). In particular, we compare the timing recorded for a calculation performed on a single node of the **MARCONI-KNL**@CINECA with the ones obtained on heterogeneous systems based on GPU accelerators, manely **Piz Daint**@CSCS (XC50 partition, nodes equipped with NVIDIA Tesla P100 GPUs), **Galileo**@CINECA (node with NVIDIA Tesla V100 GPUs), and a local cluster (**Corvina**) equipped with Intel chips and NVIDIA TITAN V cards. A more detailed technical description of the different machines is provided in Ta-

ble 6. The simulations on MARCONI-KNL and Galileo CPU-only have been performed by compiling YAMBO with the Intel 2018 compiler while the PGI compiler (supporting CUDA-FORTRAN) v19.x has been adopted for the simulations performed on hybrid architectures. For all the considered systems, simulations have been performed running on a single node and, for the hybrid systems, on a single GPU card.

Two main results emerge from the analysis of the timing reported in Table 7. The first one concerns the performance obtained by YAMBO on the CPU. In this case the best results are obtained when the code runs on Marconi@KNL (intel 2018 compiler) and Galileo (ifort compiler) as a direct consequence of both the many-core architecture of the processors and of the use of the Intel compilers and MKL linear algebra libraries. The second main result concerns the timing recorded for runs performed on the GPU cards, which point out a 5 to 10$\times$ (and even higher in some cases) speedup in the time-to-solution for the ported kernels, in particular the most time consuming routines $\chi^{\mathbf{0}}$ and $\mathbf{\Sigma_c}$, as well as for the wall-time. Most importantly, these data show an excellent portability of a complete GW calculation as performed by YAMBO on heterogeneous systems based on GPU cards, independently on the system architecture.

3.4 BigDFT

3.4.1 Poisson solver library

GPU accelerated Poisson Solver library has been deployed on the recently acquired Jean Zay supercomputer, at the Institute for Development and Resources in Intensive Scientific Computing (IDRIS), the major CNRS supercomputing center. Jean Zay's GPU partition is composed of 261 nodes, each one with 2*20 core CPUs and 4 NVIDIA V100 GPUs, with Omni-Path networking, and NVlink for connecting GPUs. Our study was part of a "great challenge" study, launched for the opening of the supercomputer to tackle large scale problems, perform difficult computations, and stretch the limits of the platform, in order to showcase its performance and also expose potential issues. For this challenge, the target is to use the PBE0 hybrid functional of BIGDFT at large scale to compute Cadmium band gap in Cadmium Telluride, in systems 8 times more computationally costly than previously performed. The target system was 216 atoms, and the main goal was to classify the different possible geometries (with or without tellurium hybridization) by energies for several charge states, in order to compare results with previously computed data for smaller systems.

GPU to GPU communication is a critical performance bottleneck in most multi-GPU codes. The Poisson Solver library developed as part of BIGDFT is able to use GPUDirect RDMA communications to improve considerably the performance of computations. In fact in this case, GPUs communicate directly with each other without performing costly data transfers. GPUDirect in Jean Zay was setup and fixed several times during the great challenge phase, allowing us to vastly improve computation times for our systems.

For benchmarking and performance tuning purposes, an H2O system with 96 atoms was used. GPU results showed that without GPUDirect, single node performance of exact-exchange on a 4-GPU run was just twice as efficient as the corresponding CPU run, as communication costs between GPUs were preventing further improvements. With GPUDirect setup, this computation is now 20 times faster than the CPU variant. This particular system actually becomes too small for multi-nodes runs when GPUDirect is

activated, as scalability after 2 nodes is absent. This is not an issue, as the GPU part of the computation has become negligible, and 128 orbitals PBE0 computations can now be performed on a single node. Larger test systems are being run in order to provide additional results.

3.4.2 Performance portability: BOAST generation of libconv sources

Libconv is a general convolution library meant to replace all convolutions in BigDFT and be also provided as a standalone, autotunable, highly optimized version for other developments to use. Autotuning strategies, used to select the fastest version of each convolution kernel (according to its parameters and the underlying platform capacities) are the core of the library, using BOAST domain specific language to express several types of common convolutions, and generating thousands of versions of each one, which are then benchmarked and selected for each platform. The use of BOAST allows the end user and developers not to focus on tedious and potentially platform-specific optimizations such as loop unrolling, efficient vectorization, intrinsics support, dimensions permutations, etc. Supported platforms range from all Intel platforms with MMX, SSE or AVX extensions, or KNL's AVX512 extensions to 64 bits ARM CPUs with NEON extensions, offering high performance at a fraction of the development cost. BOAST can also export to GPU targets with OpenCL support, and will support CUDA in the near future. We plan on exploring these capacities with libconv as well. Drawback from this technique is that initial generation of the library can be long, as thousands of variants will be tried out. But the generation is only performed once per system, and libconv will be provided with a set of already tuned variants for generic platform types. Current state of implementation is testing before integration in the BIGDFT code. The library is mainly finished, and a small subset of the BIGDFT code has already been ported to the new interface, and is being used as a prototype for integration and testing on various platforms. Testing is being performed on various platforms in parallel, with Intel x86 systems, the KNL system Marconi at Cineca, or ARM boards with Cortex A53 cores, to check correctness of the results and assess performance of the resulting implementation.

3.5 Siesta

In Siesta, the performance-portability is almost completely linked to the use of appropriate external libraries, as the lion's share of the CPU time spent by the program is associated to the solver part. As explained in the report for the D1.2 deliverable, we have considerably extended the performance enhancement possibilities of the code by the completion of the interface to the ELSI library of solvers. The performance enhancements come in three significant fronts:

- Further levels of parallelisation: A feature common in principle to all solvers is that the SIESTA-ELSI interface is fully parallelised over k-points and spins (no support yet for non-collinear spin). This means that these calculations can use two extra levels of parallelisation (beyond the standard one of parallelisation over orbitals and real-space grid), see eg Fig. 3. In addition, the PEXSI solver, beyond a reduced scaling (at most $O(N^2)$ for dense systems, and $O(N)$ for quasi-one-dimensional systems) offers *two* extra levels of parallelisation: over poles, and over trial points

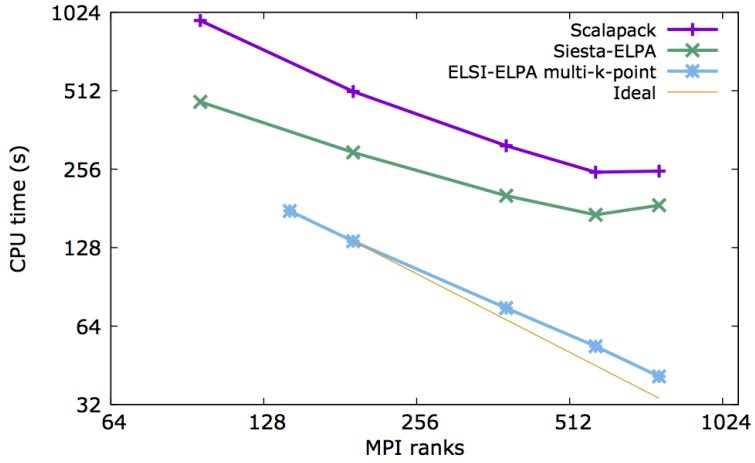

Figure 3: Performance improvement from the use of the extra level of parallelisation over k-points in Siesta using the ELSI interface with the ELPA solver, compared to the previous diagonalisation scheme (using both the standard Scalapack solver and the existing ELPA interface in Siesta). The system is bulk Si with H impurities, with 1040 atoms, 13328 orbitals, and a sampling of 8 k-points. The multi-k scheme is able to stay closer to ideal scalability for larger numbers of MPI processes.

for chemical-potential bracketing. It can be used for large systems with very high numbers of processors.

- Mixed-precision support: The ELPA solver can be invoked in single-precision mode, which can speed up the initial steps of the electronic self-consistent-field (scf) cycle. In fact, it has been shown that in Siesta one just needs to perform one or two final scf steps in double precision to maintain the standard level of precision. This leads to substantial CPU-time savings (see Fig. 4).

- Accelerator offloading: The ELPA library now offers GPU support in some kernels, and further work in the ELSI project is expanding it to more kernels. It also offers an interface to the accelerator-enabled MAGMA library. Finally, the PEXSI developers are working on adding GPU support.

  As reported earlier, there are ongoing efforts at BSC on tuning issues of MPI-GPU offloading interoperability using low-level diagonalisation libraries for Siesta. The new ELSI-related developments are more portable and already proven in pre-exascale machines such as Summit at Oak Ridge National Laboratory in the USA. We will then involve BSC in benchmarking and profiling the new functionality.

3.6 CP2K

CP2K code fully relies on the performance of the underlying libraries, in particular DBCSR – a sparse matrix-matrix multiplication library. In this regard, the CP2K code itself was not impacted by the development. All efforts were directed towards optimisation and tuning of DBCSR (see Fig. 10). The results of the full CP2K benchmark with optimised DBCSR back end are shown on the Fig. 5. The runs were executed on the

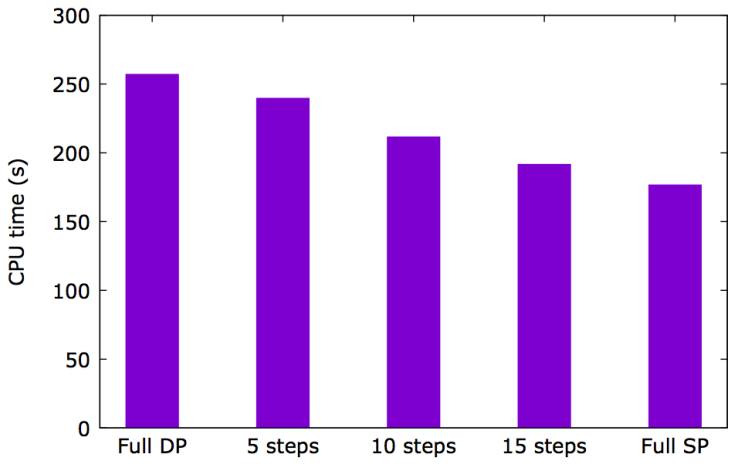

Figure 4: Performance improvement coming from the use of the mixed-precision mode in the ELPA solver. The system is a H-terminated Si quantum dot, with 1359 atoms and 14691 orbitals, and convergence of the scf cycle takes 16 steps. CPU time savings of approximately 30% can be obtained.

hybrid partition of Piz Daint supercomputer equipped with 12-core Intel Haswell CPU and NVIDIA P100 GPU card. The full benchmark description is available on the CP2K GitHub portal.[2]

4 Implementation of performance portable code

In addition to code closely tied to one of the flagship codes we also work on the implementation of performance portable libraries. Hence we here report on such activities of which all codes within MAX and also the wider community could benefit. Performance portability is one of the cornerstones of HPC today. Several GPU programming models (CUDA, ROCm, OpenACC, OpenMP and OpenCL) coexist today, but none of them is good enough for all the problems and one has to find a compromise between the difficulty of writing a GPU specific code and a gained performance.

For example, CSCS is involved in the development of several libraries for electronic structure codes: SIRIUS, DBCSR, COSMA and SpFFT. All these libraries were written in C++ with CUDA programming model. The CUDA programming model allowed for a quick migration to ROCm and now all these libraries are capable of running on NVIDIA and AMD GPU cards thus ensuring both performance and portability. In the past 12 months the following developments have been completed at CSCS:

- SIRIUS library was ported to ROCm; the eigen-solver for AMD GPUs is still missing, but this problem will be addressed by MAGMA library developers in collaboration with AMD;
- SIRIUS library was tweaked to run on multi-GPU nodes (the case of Summit supercomputer at Oak Ridge);

[2] https://github.com/cp2k/cp2k/tree/master/benchmarks/QS_DM_LS

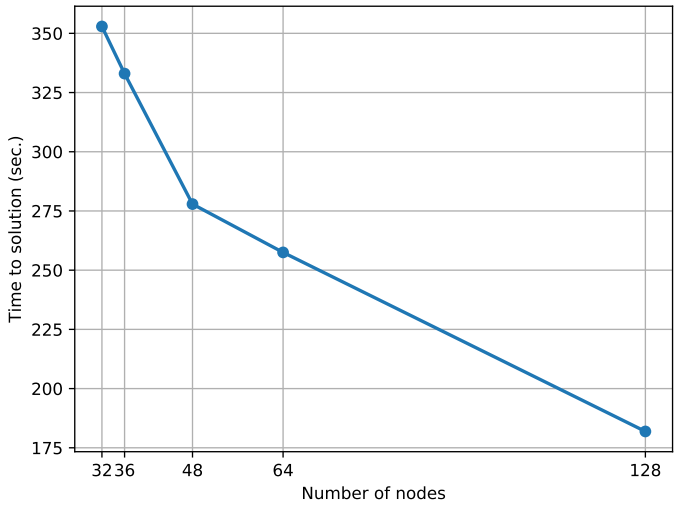

Figure 5: H_2O DFT linear scaling calculation. The unit cell contains 20736 atoms in a 59 cubic angstrom box (6912 water molecules in total). An LDA functional is used with a DZVP MOLOPT basis set and a 300 Ry cut-off. Time to solution is plotted.

- DBCSR library for sparse matrix-matrix multiplications was extend with automatically tuned GPU kernels which are compiled "on-the-fly" using CUDA's Just in time (JIT) capabilities;
- DBCSR library was ported to ROCm and JIT capabilities of ROCm were tested;
- DBCSR library was tweaked for multi-GPU node case;
- auto tuned parameters for P100, V100 and Mi50 GPU cards were added to the DBCSR repository;
- COSMA library to perform communication optimal matrix-matrix multiplication was ported to CUDA and ROCm programming models;
- `p{d,z}gemm` wrappers for COSMA library were implemented together with the data layout transformation module;
- SpFFT library to perform sparse (in frequency domain) Fourier transforms which are specific to electronic structure codes was written from scratch; the library has CUDA and ROCm backends and is already integrated into SIRIUS code; SpFFT is using GPU-friendly 1D-2D transformation decomposition;
- SpFFT library was tweaked for multi-GPU node case.

The results of benchmarks for SpFFT, COSMA and DBCSR libraries are presented in Fig.6-10.

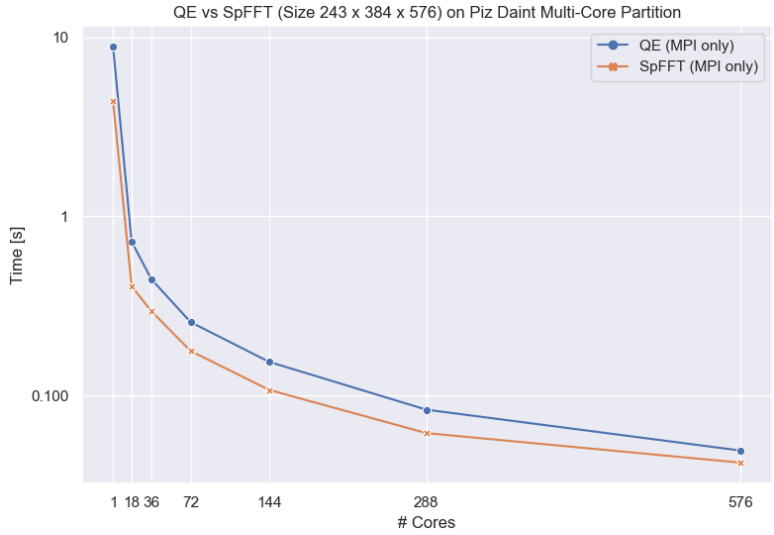

Figure 6: Strong scaling of the FFTXlib and SpFFT libraries measured with FFTXlib mini-app on 1-16 multi-core nodes of Piz Daint.

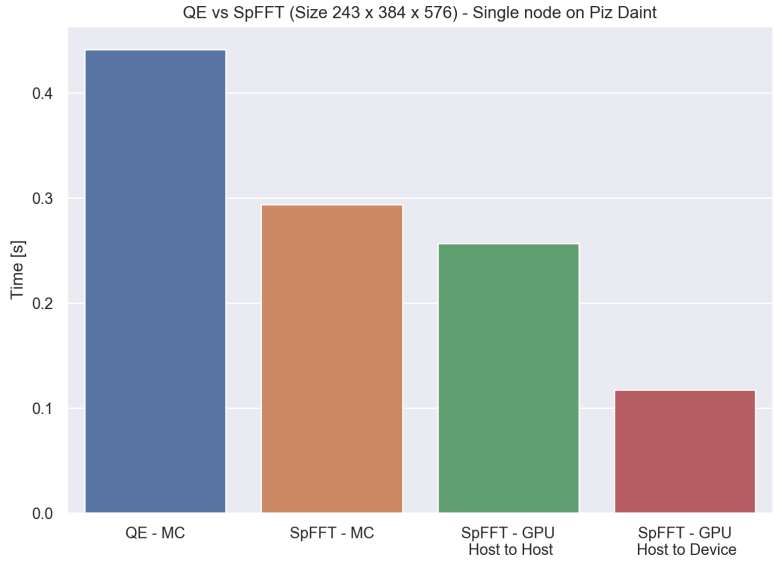

Figure 7: Performance of the FFTXlib and SpFFT on a single node. MC: dual socket multi-core node equipped with 18-core Intel Broadwells, GPU: hybrid nod containing 12-core Intel Haswell and NVIDIA P100 GPU. Host-to-host: real-space data is transferred back to the host memory, Host-to-device: real-space data remains in the GPU memory.

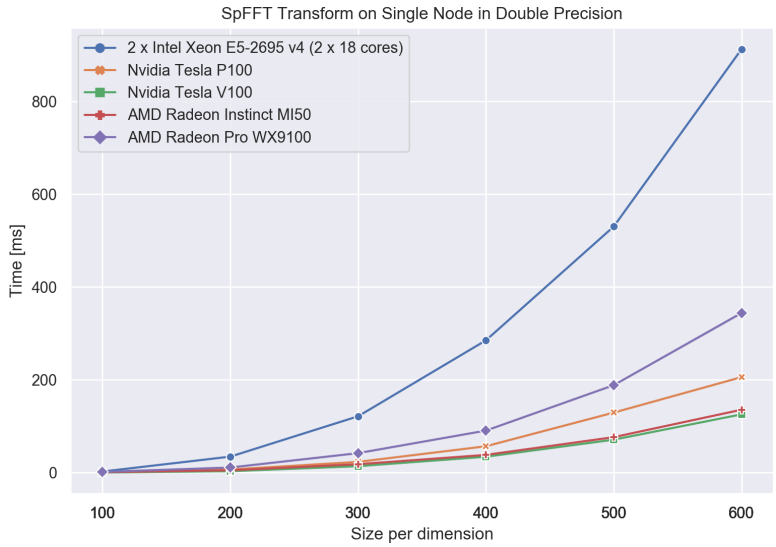

Figure 8: Execution time of SpFFT for various grid dimension sizes on Intel, NVIDIA, and AMD architectures.

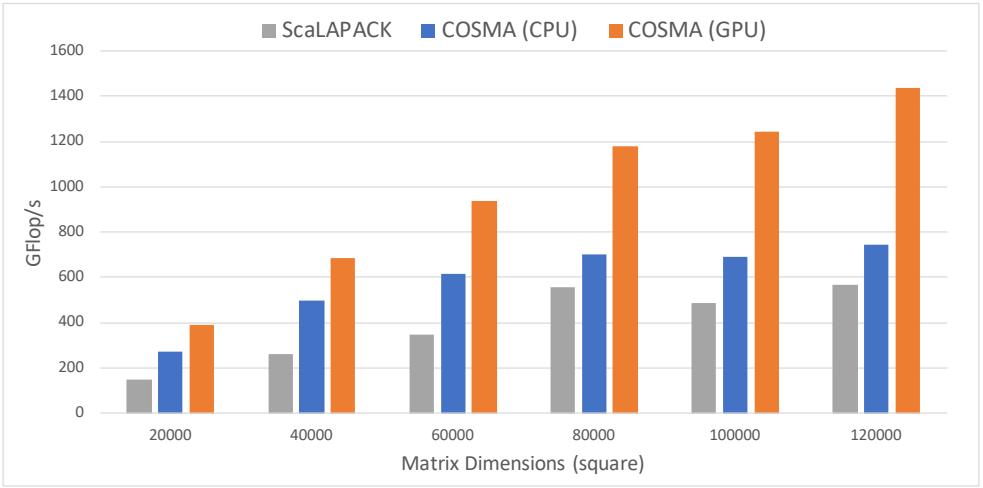

Figure 9: Performance comparison of the ScaLAPACK and COSMA libraries in the square matrix-matrix multiplication test. Performance per node is shown. The tests were run on 128 muli-core and hybrid nodes of Piz Daint.

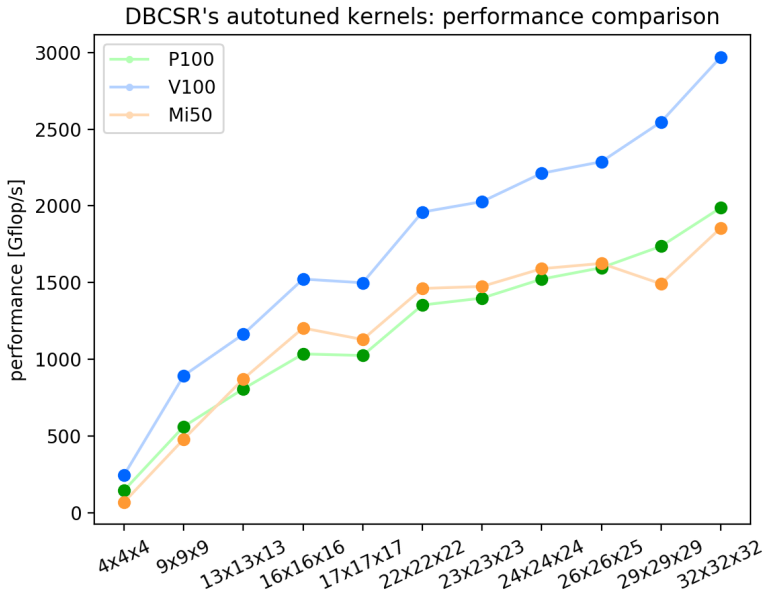

Figure 10: Performance of DBCSR matrix-matrix multiplication kernels for several m,n,k triples (x-axes) on the NVIDIA and AMD cards.

5 Performance prediction and adaptation of algorithms

In the last section of this report we focus on the performance prediction and our first efforts on auto-tuning approaches to facilitate users of our codes to perform simulations with optimal performance.

5.1 Predicting QE execution time with A.I. tools

The focus of this work was the prediction of execution times for QUANTUM ESPRESSO, based on input data only. The approach follows Artificial Intelligence methods, in which a large database of existing calculation is used as "training set" for a Machine Learning procedure, aimed towards reproducing a specified target quantity (related to execution time).

A dataset of approximately 5000 simulations, part of a screening procedure from the Crystallography Open Database, has been selected. The dataset consisted mostly of relatively small unit cells, involving a large part (72) of the elements in the periodic table. The simulations were run on 8 compute nodes of different machines mostly at CSCS. For a more detailed description of the dataset see the Materials Cloud archive.[3] About 80 % of the data was used for training, the remaining 20% for test. The time spent in routine "cbands" of PWscf was used as the target to be reproduced. This is typically the most time-consuming part of a self-consistent calculation. Three different Machine Learning algorithms were implemented: Linear Regression (LR), Kernel Ridge Regression (KRR), Fully Connected Neural Network (FCNN). The prediction model is based upon several parameters that cover all the main factors affecting execution: type of machine and of

[3] https://doi.org/10.24435/materialscloud:2017.0008/v2

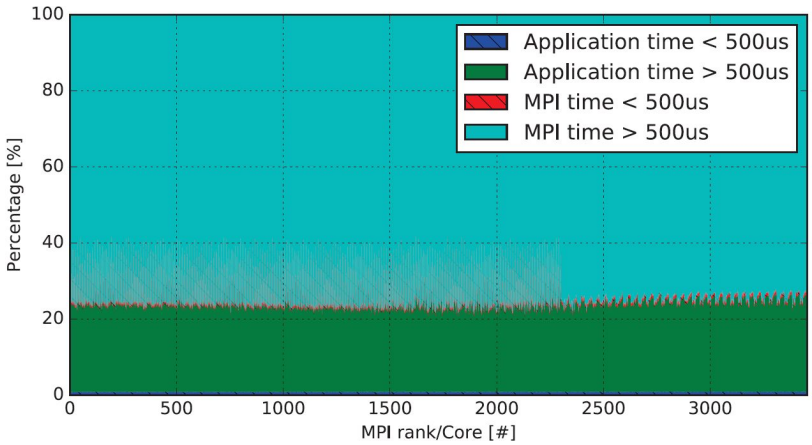

Figure 11: QUANTUM ESPRESSO communication and computations phases classified according to the length of the phase, larger or smaller than 500microsecond (500microsecond is the time it takes for the CPU core to change its clock speed).

parallelisation, diagonalisation algorithm, size of the system (i.e., unit cell and number of electrons) , basis set size.

The first results are very encouraging, especially for the FCNN model. A paper with the description of the procedure and of the results obtained has been presented at the PASC19 conference.[4] More work is ongoing on further improvements, notably: enlarging the dataset and the size of the prediction models; use a more diverse dataset with multiple code versions; extend the model to predict the number of iterations and other properties.

5.2 Profiling and auto-tuning for optimizing Total Cost of Energy of QE simulation

The hardware power management in nowadays processing elements is effective in reducing the power consumption of idle resources. However, in large-scale MPI parallel applications that fully utilize all the assigned processing elements, the logic of the hardware power management lacks a global view of all the resources the application uses, and is unable to exploit workload unbalance, synchronization, and communication slack to save energy. To remedy this problem, the QUANTUM ESPRESSO team at Cineca worked to enable a technology called COUNTDOWN [5] developed also thanks to FET-HPC project (ANTAREX). COUNTDOWN exposes the same interface of a standard MPI library and can intercept all MPI calls from the application on the one side, interacts with the hardware power manager through specific events on the other side. COUNTDOWN is endowed with profile capabilities which allow a detailed analysis of the application performance (fig. 11) and monitor the energy/power consumed by the CPU and memory.

Based on these measurements, COUNTDOWN is able to change the frequency of single cores within a CPU in application phases not requiring heavy CPU processing (e.g. during an MPI barrier), and saving a significant amount of energy and reducing

[4] https://dl.acm.org/citation.cfm?doid=3324989.3325720
[5] https://github.com/EEESlab/countdown

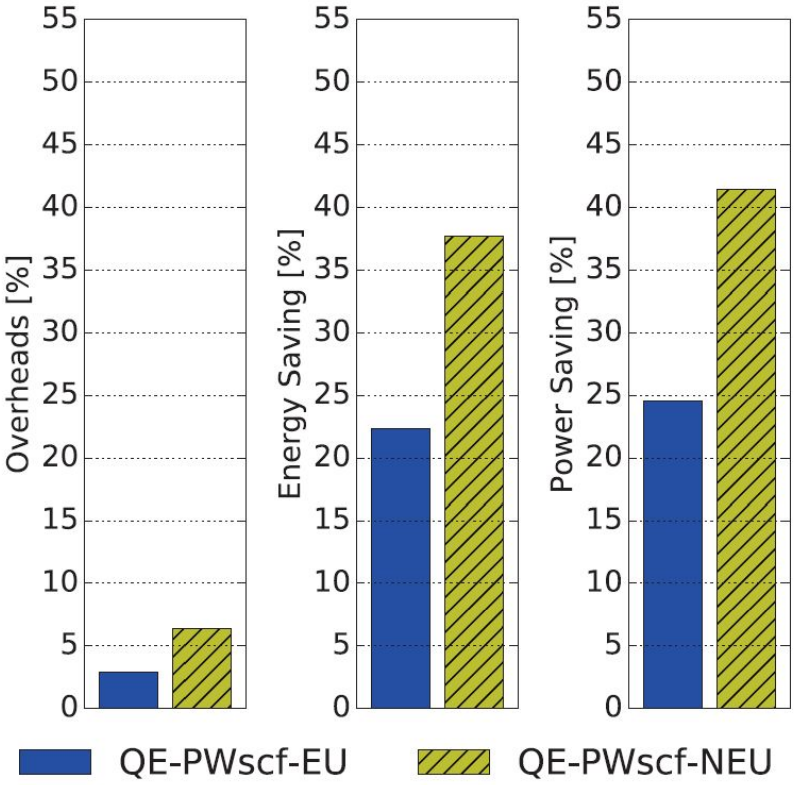

Figure 12: Energy savings and overhead for two different value of "ndiag" linear-algebra parallelisation parameter: EU optimal ndiag, NEU suboptimal ndiag.

the TCO (Total Cost of Energy) of the simulation. COUNTDOWN requires privileged access to power management module of the linux kernel, that can be managed with a specific module (msrsafe) developed by Livermore [6] to be installed in the target supercomputer. We use the Galileo PRACE Tier-1 system to perform the test and validation of the COUNTDOWN library together with QUANTUM ESPRESSO using the msrsafe kernel module (deployed in the Galileo Tier-1 cluster). We then demonstrate that for large parallel QUANTUM ESPRESSO jobs, we can save up to 35% of the energy with a minimal penalty in term of time to solution (fig. 12). The benchmark test case reproduces a layered structure of Iridium, Cobalt and Graphene plus a molecular compound (iron-phthalocyanine) deposited on top [4]. The whole simulation box includes 662 atoms and 3662 Kohn-Sham states. The total number of plane waves exceeds one million. During the execution, the main memory occupation may peak at 2 to 6 terabytes depending upon the selected parallelisation parameters.

As a conclusion of this proof of concept, Cineca plans to deploy this technology on the Tier-0 system Marconi and the future EuroHPC system Leonardo. We then suggest to have this energy consumption auto-tuning option for the future European exascale systems, for all those applications that will be validated and enabled. It will be always a user decision if a specific run should be executed with the power management activated.

[6] https://github.com/LLNL/msr-safe

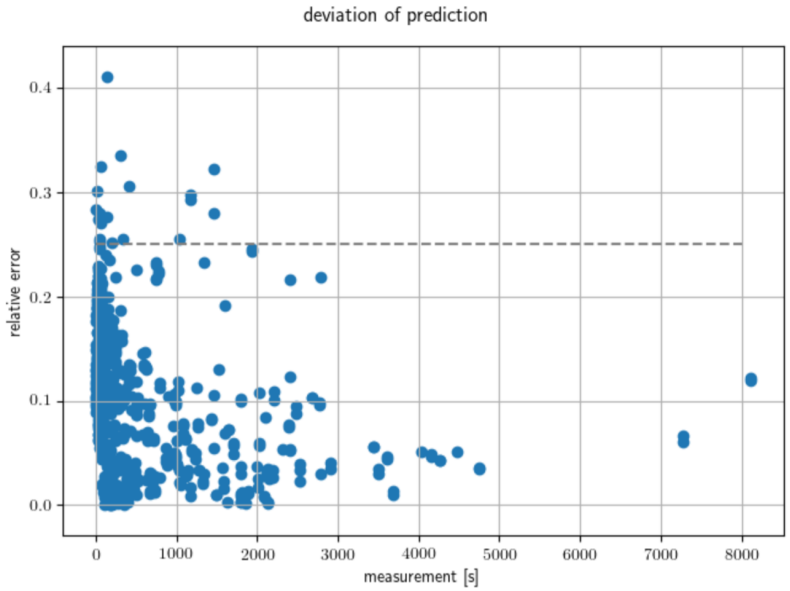

Figure 13: FLEUR model performance. The x-axis shows the measured runtime while the y-axis shows relative error that the prediction causes. The horizontal line marks a deviation of 25%.

5.3 Performance prediction of FLEUR

A performance model of FLEUR able to predict the sequential runtime of a single FLEUR iteration cycle has been created. A mix approach was followed while establishing the model: on the one hand knowledge about the algorithms that FLEUR uses is taken into account. On the other hand the code is partially considered as being a black box and runtime measurements are analysed in order to find the influences on the runtime that were not found by the first approach. The parameters of the model are the number of plane waves, the number of atoms, the number of atom types, the number of valence electrons, the number of local orbitals, the number of spins and whether the system is real or complex. Since different parts of the code (i.e. diagonalisation, spherical and non-spherical matrix setup, etc.) have different scaling behaviour, the model for the whole self-consistency cycle is a sum of particular models for those parts.

Figure 13 shows the relative error in the prediction in relation to the actual runtime measurement. While starting with a very high error from the measured runtime the error decreases very fast with increasing runtime measurement. No computation has a relative error of more than 50% and 96% of the data have a lower relative error than 25%. The model can be adapted to new hardware etc. by measuring the runtime of well-chosen training computations that were done on the new setup.

5.4 Performance prediction of BigDFT

The SimGrid framework [5] is a framework for developing simulators of distributed applications used to prototype, evaluate and compare relevant platform configurations, system designs, and algorithmic approaches. This versatile scientific instrument has been used for simulation studies in Grid Computing, Cloud Computing, HPC, Volunteer Com-

puting and P2P Systems. It has also been shown to be both more realistic and more scalable than its major competitors, thus lowering the boundaries between research domains.

BIGDFT has already been partly ported on top of SimGrid and very promising simulation results have been obtained [6]. The SMPI interface provided by SimGrid can be used to replace any MPI library use in HPC codes at a very low cost, and simulate the behaviour of the actual application on any potential platform. In the simplest case, computation parts will be executed, benchmarked, and the resulted timings will be adapted and injected on the simulated environment. This means that results will be preserved, and this is necessary for codes with data-dependent behaviour. But this is also expensive both in time and memory, as simulation is performed on a single workstation/computational node.

SMPI provides ways to overcome such limitations, and replace repeated computational parts with none to minimal interference in the code. Large memory allocations can be folded into smaller ones, shared between processes, in order to overcome memory size limitations. Data is then rendered incorrect, but computation is still performed. This allows to simulate HPC-sized systems on a personal computer. This can be done either by flagging allocations that can be folded (to avoid folding control data that are necessary for the execution and communication), or just by setting a threshold of size, above which all allocations will be folded. In BIGDFT, the allocations are performed through the futile library, which handles SimGrid shared allocations easily, allowing fine tuning of such behaviour.

For computationally expensive parts, kernels can be emulated instead of computed. The classical SMPI way of handling these parts is adding C sampling macros over existing loops, which will perform a sampling phase for the first iterations. In this phase, the code is executed and benchmarked until a certain stability is reached. Then, the remaining iterations are skipped, integrating the time in the simulation, instead of performing the computations. This is not easily done in Fortran, as it relies heavily on the C pre-processor. Another drawback is that the computed time that is integrated inside the simulation is always the same for each subsequent call to the sampled kernel. In reality, variability is a big part of computation, and a better accuracy is reached when this variability is taken into account.

In this regard, we propose a novel approach for modelling the convolutions of BIGDFT. Leveraging the modularity introduced by the usage of specialised libraries and the introspection capabilities of the libconv library, a "libsimconv" variant of this library can be generated easily from its BOAST representation. Indeed the convolution library already computes and exposes the computational cost of each one of its internal operations and is able to provide this information through a query mode. Adding a simulation mode is hence easily performed, silently replacing the computation of the convolution by an equivalent injection of time. Integrating the logic in the library will also allow to integrate reproducible noise models, to not only integrate a single amount of flops in the timeline, but a realistically randomised one.

This work is currently performed within SimGrid to provide a simulated library which would realistically emulate BLAS functions and could be linked instead of any BLAS library for codes relying on it. For these, modelling is rather tedious, and results depends on a huge number of parameters. A complex calibration phase is hence needed

to provide accurate yet simple performance models. Data measured on real-life platforms is being fitted generically through Bayesian sampling with the Stan analysis tools, allowing to infer such models programmatically and generate several realistic models for each function. These various models can be used in the simulated library and provide another level of realistic randomisation. For instance, each simulated processor could use a slightly different model for each kernel, thus emulating real-life platform behaviour, with some processors being slightly faster or slower than others.

With these tools, the performance prediction of codes such as BIGDFT can be both accurate enough and fast enough to provide important information for developers. In the case of BIGDFT, this would for instance translate in a tool to provide insight on which parameters will yield fastest results on a given supercomputer before submitting any real job on it. For example, studying the point at which linear scaling BIGDFT becomes faster than cubic BIGDFT is an important goal to achieve.

6 Conclusions and perspectives

All the MAX flagship codes demonstrate their ability to run on a diverse set of different hardware and software environments and in many cases the performance was already significantly improved in the course of the project. A lot of attention has been devoted to the porting to GPU-based accelerators and all codes can report improvements on their performance on such systems. Importantly, fully fledged DFT and GW workflows can be now performed on heterogeneous systems by using MAX codes. Moreover, the first experience on ARM based HPC architectures has currently been obtained and performance portability issues to this specific hardware are being identified.

The future work within WP2 will focus on efforts to unify the different approaches and to further extend the supported architectures. A particular focus will be put on the challenges imposed by accelerator architectures beyond the NVIDIA/CUDA framework. Here we foresee an increase in activities using OpenMP 5.0 and other new programming paradigms. Profiting from the modularisation work in WP1 and actually in close synergy we also plan to continue working on providing performance portability to the libraries developed.

References

[1] Cavazzoni, C. *et al.* First report on code profiling and bottleneck identification, structured plan of forward activities. Deliverable D4.2 of the H2020 CoE MaX (final version as of 30/06/2019). EC grant agreement no: 824143, CINECA, Bologna, Italy (2019).

[2] Genovese, L. *et al.* First release of MAX software: report on the identified actions, update of the software development plan, and software release. Deliverable D3.2 of the H2020 CoE MaX (final version as of 30/11/2019). EC grant agreement no: 824143, CEA, France. (2019).

[3] Denk, R. *et al.* Exciton-dominated optical response of ultra-narrow graphene nanoribbons. *Nat. Commun.* **5**, 4253 (2014).

[4] Avvisati, G. *et al.* Orbital symmetry driven ferromagnetic and antiferromagnetic coupling of molecular systems. *Nano Lett.* **18**, 2268–2273 (2018).

[5] Casanova, H., Legrand, A. & Quinson, M. Simgrid: A generic framework for large-scale distributed experiments. In *Tenth International Conference on Computer Modeling and Simulation (uksim 2008)*, 126–131 (2008).

[6] Bédaride, P. *et al.* Toward better simulation of MPI applications on ethernet/TCP networks. In Jarvis, S. A., Wright, S. A. & Hammond, S. D. (eds.) *High Performance Computing Systems. Performance Modeling, Benchmarking and Simulation*, 158–181 (Springer International Publishing, Cham, 2014).