D4.6

Final report on co-design activities

Daniele Cesarini, Andrea Piserchia, Conrad Hillairet,
Nicola Spallanzani, Andreas Beckmann, Anton Kozhevnikov,
Augustin Degomme, Alberto García, and Fabio Affinito

Due date of deliverable: 30/11/2021
Actual submission date: 07/12/2021

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

Document information

Project acronym:	MaX
Project full title:	Materials Design at the Exascale
Research Action Project type:	European Centre of Excellence in materials modelling, simulations and design
EC Grant agreement no.:	824143
Project starting / end date:	01/12/2018 (month 1) / 31/05/2022 (month 42)
Website:	www.max-centre.eu
Deliverable No.:	D4.6

Authors: D. Cesarini, A. Piserchia, C. Hillairet, N. Spallanzani, A. Beckmann, A. Kozhevnikov, A. Degomme, A. García, and F. Affinito

To be cited as: D. Cesarini et al., (2021): Final report on co-design activities. Deliverable D4.6 of the H2020 project MaX (final version as of 07/12/2021). EC grant agreement no: 824143, CINECA, Casalecchio di Reno (BO), Italy.

Disclaimer:

D4.6 Final report on co-design activities

Content

1. Executive Summary

In this Deliverable we report on the co-design actions put in place by the MaX CoE in the past 18 months (Jun 2020 - Nov 2021). We have focused on four main activities: (1) we studied the performance of MaX applications on selected HPC micro-architectures relevant for the EuroHPC context; (2) importantly, in collaboration with ARM we also tested selected MaX codes on the ARM Neoverse micro-architecture, relevant, e.g., for the future Sipearl Rhea and Nvidia GRACE; (3) we have made experience with the MFAT tools to address memory allocations and usage, important e.g. for the exploitation of high-bandwidth memory; (4) we have tested and experimented different programming models to be used in the context of performance portability, with a special emphasis on heterogeneous architectures. This also led to the design of domain-specific libraries, as discussed in previous deliverables and in the activity of WP2.

Besides directly targeting the performance of MaX codes on relevant (micro-)architectures, and characterizing the computational kernels of interest, we have also gathered a collection of data related to MaX codes on different HW architectures, which allows one to correlated the performance of MaX codes, and of the dominant implemented algorithms, with the features of the HW. On the one hand, this allows us to guide the evolution of the MaX codes towards the optimal exploitation of the HPC infrastructures and, on the other, to provide the HW designer with a large amount of information in order to take trade-off decisions when realizing or integrating new products. The design of the next exascale systems will be strongly bound to the energetic impact as one of its main constraints: for this reason, power efficiency is now considered as one of the relevant metrics in comparing the different architectures.

In the first part of this document we report an analysis on the performance of the MaX codes. Here we focus the investigation on the CPU exploitation by comparing 4 different microarchitectures: Intel Xeon Skylake, ARM ThunderX2, AMD Rome, and IBM Power9. For each of those, we performed an analysis based on the top-down microarchitecture analysis method (TMAM), and on the roofline model. Moreover, we compared the last version of the MaX flagship codes with a baseline referred to their version at the starting point of MaX (phase 2) using a set of six metrics of interest (execution time, IPC, vector ratio, MPI time, Flop/s, energy efficiency).

In the second part of this Deliverable we focus on some ARM-based architectures. Thanks to our partner ARM, we had the opportunity to examine the performance of some of the MaX flagship applications on the ARM Neoverse cores. This complements the analysis performed in the previous section on Thunder X2 and puts the basis to understand the behaviour of MaX codes on future architectures relevant for the EU HPC ecosystem, such as those based on Sipearl Rhea and Nvidia GRACE (as for ALPS, the next CSCS supercomputer). In doing this, the data on the Neoverse N1 chip are compared with the Intel Xeon Skylake processors, for selected MaX flagship codes (Yambo and BigDFT). BigDFT is also one of the codes involved in EUPEX, the first EuroHPC Advanced Pilot for Exascale, aimed at leading to the realization of the European processor, as an outcome of the EPI project.

Next, we describe and characterize the MFAT tools, used to track the allocation of the memory pages during the execution of an application. This is of crucial importance to understand the exploitation of the memory hierarchies and, in particular, the High-Bandwidth memories (HBM).

Finally, the last part of this deliverable is focused on the proof-of-concept (PoC) of programming paradigms and domain-specific libraries. As we discussed in previous WP4 deliverables, these PoCs arise from the need of making it easier to run on new (heterogenous or long-vector) architectures with minimum coding effort. We kept on exploring programming paradigms such as OpenMP and OpenACC for a directive-based approach, but also, by adopting the concept of "separation of concerns", we pursued the approach of domain-specific libraries. In this deliverable, more specifically, we discuss the use of libraries based on HPX for tall-and-skinny matrix multiplications and of *libconv* for exploiting the SVE in the convolution kernel of BigDFT.

2. Performance Portability and Energy Efficiency of MaX Codes

Within this co-design activity, the performance and the power efficiency of all MaX codes have been extensively analyzed in order to quantify the impact of the project work on the evolution of MaX codes. To complete this analysis, we compared the codes using the latest MaX releases with a version of the code dating back to the start of the project. We here show the evolution in terms of execution time, microarchitecture performance, and power efficiency covering most of the architectures used in today's supercomputers.

This analysis mainly focuses on single-node configuration using general-purpose CPUs, since most of the MaX codes would not support accelerated systems when this project started. A more extensive analysis for the heterogeneous architectures and power efficiency will be appropriate when multiple accelerated technologies are available in large-scale production systems. To conclude, we present an analysis on one of the MaX flagship codes, Quantum ESPRESSO, run at scale in order to explore how to tune power efficiency using different MPI and OpenMP configurations.

2.1. Performance Models and Metrics

We start our analysis by showing the metrics and the performance models that we have used to characterize the performance and the efficiency of the MaX flagship codes.

2.1.1. Performance and Microarchitecture Efficiency

The first performance metric that we take into consideration is the **execution time**. By using this metric it is possible to easily compare the application performance among different HPC systems. It is a common metric used in scientific applications in order to identify the most suitable HPC system for the best time-to-science performance. This metric is widely recognized as the best one to characterize the overall performance of an application on different systems, even though it provides no information on the microarchitecture efficiency of the run. For instance, if we run an application on an HPC system and, sequentially, we run the same application on a different HPC system, we can compare both execution times to find out which is the fastest machine. If the application on the first HPC system requires 100 seconds to conclude and, on the second system, requires 75 seconds to conclude, we can say that the second system is 25% faster than the first one. But if the second system has twice the number of cores and an operating frequency 30% higher, this means that the application efficiency of the second system is drastically lower than the first one.

An HPC application is usually a representation of an algorithm that solves a specific scientific problem, defined as a finite sequence of well-defined operations. In the computer science world, a scientific HPC application is represented as a set of finite sequences of architectural instructions that a computer needs to perform in order to conclude the computation. The higher is the number of instructions executed each second, the shorter will be the execution time. In these terms, maintaining a large number of instructions executed every second is crucial to reach the best application performance. Today's CPUs are able to parallelize the execution of the instructions to maximize the throughput of the system, a capability called instruction-level parallelism - the metric to measure the level of parallelism - and is typically represented as the number of **instructions executed for each cycle (IPC)**. Applications that are able to maintain a high level of IPC usually show better

performance as they require a shorter execution time. If the IPC of the application is close to the theoretical IPC of the system, it can be said that the application uses the system efficiently.

IPC is a metric that represents the overall utilization of the microarchitecture, and it can be a misleading performance metric because the application can reach high IPC by using useless or little/less relevant instructions. For instance, an application can emit a lot of scalar instructions to perform a scientific workload, whereas it would be better to execute high efficient vector instructions which, with a single instruction, can perform in parallel multiple arithmetic operations. In this case, even with a low IPC, the application can have a lower execution time.

In these terms, a more efficient metric would be **the number of floating-point operations per second (FLOPs) delivered** from the CPU. A large number of FLOPs shows the good use of the microarchitecture of the system: this is the metric used in the Top500[1] list to classify the fastest supercomputer in the world. Usually, HPC applications need to solve arithmetic-based problems, so counting FLOPs is a valuable metric to measure the overall efficiency of a system. While this metric is crucial in the HPC domain, we cannot limit our performance analysis to this single metric to evaluate the efficiency of the application. While FLOPs only consider arithmetic operations executed by the CPU, algorithms of scientific applications can be very complex and, from an architectural point of view, the application can not be reduced only to a subset of arithmetic operations to perform as they need to leverage on non-arithmetic instructions as well (for instance, moving data from the memory or computing the branch path of an algorithm). This also means that it is impossible to run at the peak FLOPs of a CPU as HPC applications are also composed of non-arithmetic instructions which often dominate the computation. It is also important to remember that a large number of FLOPs reduces the IPC due to a greater use of the memory bandwidth which usually induces memory stalls. Furthermore, we need to remember that FLOPs are not a good metric when it is important to analyze general purpose applications, such as web servers, application servers, database management systems, etc.

Moreover, it is a few years that the race of hardware designers to reach the highest FLOPs at any cost is no longer sustainable for the continuous shirking of the transistors that consequently induce higher power density and consumption in CPUs. In today's supercomputers the new performance race has therefore shifted from the highest FLOPs to the highest **power efficiency, usually expressed as FLOPs per Watt (FLOPs/W)**. As of today, the energy-efficient metric is becoming the dominant to express the overall efficiency of a CPU, and it must be considered in relation to the absolute value of FLOPs to evaluate the overall efficiency of a computational system.

To conclude, a good performance analysis must take into account all the above metrics (and possibly even more), and also consider that:

- It is impossible to reach the theoretical peak performance of a system;
- A single performance metric can limit other performance metrics that we are observing (trade-off problem);
- A single performance metric cannot express the overall efficiency of a microarchitecture but we need to take into account multiple metrics.

[1] https://www.top500.org/

2.1.2. Top-down Microarchitecture Analysis

A modern CPU employs pipelining as well as other techniques, like hardware threading, out-of-order execution, and instruction-level parallelism, to utilize resources as much effectively as possible. The top-down microarchitecture analysis method (TMAM) is used to identify dominant performance bottlenecks in an application. Its purpose is to show, on average, how well the CPU's pipeline(s) are being utilized while running an application. As detailed in the architecture chapters, though modern CPU's pipelines are quite complex, it is possible to divide them conceptually into two halves:

- **Front-end**: is responsible for fetching architectural instructions (Mops) and decoding them into microinstructions (µops) specific for that CPU. The µops are often allocated in a queue ready to be executed from the back-end.
- **Back-end**: is responsible for fetching and executing the µops using the available execution units of the CPU. It is also responsible for fetching data from the memory that µops require to process.

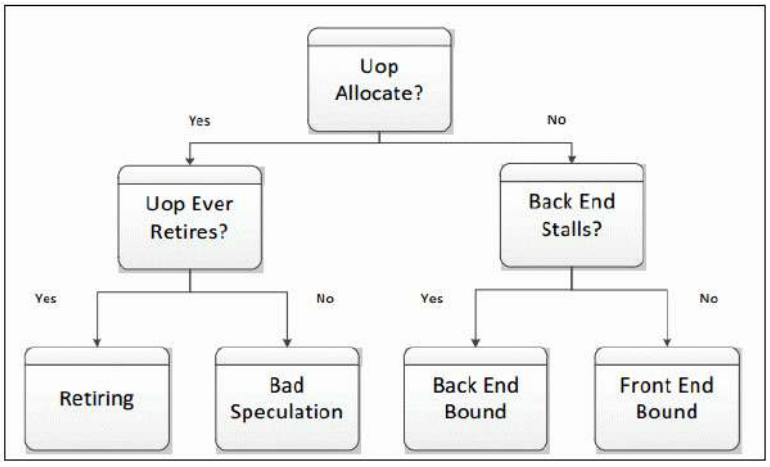

Fig. 1: Breakdown of TMAM reasons to classify if cycles are used to retired instructions or stalling[2].

In this methodology, we try to detect what stalls our execution starting from the high-level components and narrowing down the source of performance inefficiencies. Fig. 1 shows a hierarchical representation of questions to understand what happens in a pipelined CPU. The first question asks if the µops are executing (retiring) or stalling in the pipeline. If the cycles spent from the CPU are retiring instructions, we have to identify if the instructions have been actually completed or only speculated without retiring them. This is because a CPU often executes instructions without knowing if these will be needed (for instance, during an algorithmic branch). Instead, if the pipeline is stalling we have to identify the reason. The stalls can be caused by the front-end or by the back-end of the core. Front-end issues denote an under-supply of the back-end, usually due to a latency of the instruction fetching or bandwidth overloading of the instruction cache. While back-end issues usually reflect problems in lack of required resources for accepting µops in the backend. Examples of back-end issues include memory access bottlenecks like bandwidth or latency and can be caused by the execution units being overloaded.

[2] https://easyperf.net/blog/2019/02/09/Top-Down-performance-analysis-methodology

In Table 1, we show general guidelines for a good mix of cycles spent on each category for different kinds of applications.

Category	**Client/Desktop Application**	**Server/Database/Distributed application**	**HPC application (target)**
Retiring	20-50%	10-30%	30-70%
Back-End Bound	20-40%	20-60%	*20-40%*
Front-End Bound	5-10%	10-25%	5-10%
Bad Speculation	5-10%	5-10%	1-5%

Table 1: Guidelines for a good mix of cycles spent on each category for different kinds of applications.

In our analysis, we use TMAM methodology in order to classify the MaX codes by exploring the main bottlenecks and the microarchitecture efficiency level of each application. Unfortunately, we had to restrict this methodology on Intel Skylake and Cavium ThunderX2 CPUs, as these are the only ones that support the performance counters to extrapolate a TMAM analysis.

2.1.3. Roofline model

The second performance model we employed in this analysis is the roofline model[3], an intuitive model used to estimate the compute-memory ratio of a kernel or an application and hence to quickly identify if the computation is compute or memory bound. The roofline model correlates FLOPs and memory bandwidth in a two-dimensional plot with FLOPs in the Y-axis and operational intensity (FLOPs/Byte) in the X-axis as shown in Fig. 2.

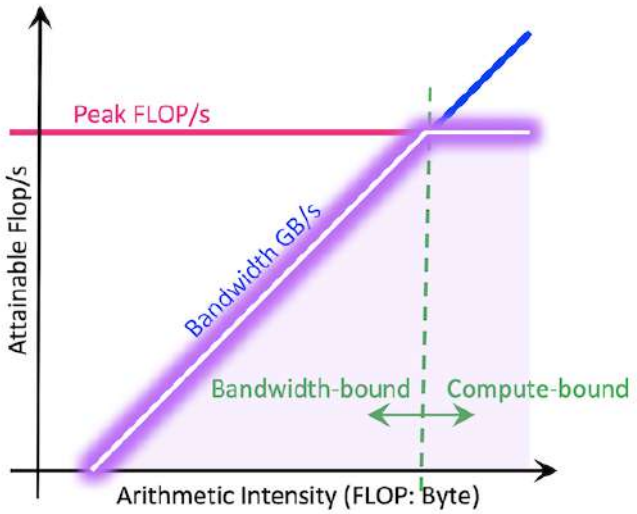

Fig. 2: Overview of roofline model[4].

[3] W. Samuel, A. Waterman, and D. Patterson. "Roofline: an insightful visual performance model for multicore architectures." *Communications of the ACM* 52.4 (2009): 65-76.
[4] https://docs.nersc.gov/tools/performance/roofline/

The operational intensity is the ratio between the FLOPs performed by an application and the total data movement required to deliver such FLOPs. The ridge point of the roofline is called "machine balance" point: usually, application codes lower than this point are considered to be bandwidth bound while, on the other hand, codes higher than this point are considered to be compute bound.

This roofline model has been used to classify the MaX codes even though, unfortunately, we had to restrict the roofline analysis only to the IBM Power9 system as it is the only system that monitors the memory bandwidth (necessary to plot the roofline) due to some kernel limitations on the other target systems.

2.2. Benchmarks and software stack

This co-design analysis has involved all MaX flagship codes in two different versions. The first one represents the latest revision from MaX activities, while the second one is a revision of the code before the starting date of the MaX project. We have run all versions on four architectures used in real-production HPC systems, and have collected microarchitectural performance metrics to evaluate the impact of MaX activities on each code.

2.2.1. Benchmark characteristics and datasets inputs

For each application, we have identified an input dataset to fit the application instance in a single node. We selected a small scientific case, big enough to respect the following resource occupancy of the node to have a fully loaded system:

- **CPU**: we have run all the codes using all the computing resources available in the node. This also means that we have instantiated a number of MPI processes equal to the number of cores available in the node. We haven't used the simultaneous multithreading (SMT) technology, a technique for improving the overall efficiency of superscalar CPUs with hardware multithreading, as it is often disabled by default in many HPC systems for the excessive overhead in the core front-end when the application workload is bound on the execution units. In our analysis, we make an exception in disabling the SMT technology only for a specific CPU built to push at an extreme level the instruction-level parallelism. We have disabled the OpenMP capability of the application: though this programming model allows for a better use of memory and reduced network communication, it also induces a worse computing utilization, so it doesn't usually pay off its utilization in a single-node run but it is often used at scale (see the Paragraph 2.9 on the MPI and OpenMP performance at scale). However, this is not always the case, for instance: e.g., the BigDFT application uses a configuration that involves the OpenMP acceleration due to the parallelism being constrained to physics elements of the simulation, so without OpenMP it would not be possible to utilize all computing elements of the system.
- **DRAM**: All the dataset inputs that we used in this analysis require tens to hundreds of Gigabytes of DRAM utilization. From the microarchitectural point of view, this is enough to saturate the last level cache of the node to show memory behaviour when large dataset inputs are used, a common case in material science codes.
- **Network**: The MPI processes leverage the shared memory optimization implemented in MPI libraries, so no inter-node communication is involved during the benchmarking.

- **Storage**: The impact of the I/O communication for these such small inputs is negligible due to the fact that I/O is a challenging problem when the applications run at scale.

We run the MaX codes by choosing specific input datasets from the publicly available MaX benchmark repository[5]:

- **Quantum ESPRESSO:** the input dataset performs a self-consistent field (SCF) calculation using pw.x executable of a system constituted by 443 atoms per cell of carbon and iridium (2987 electrons, 30 Ry of kinetic energy cutoff). The DFT calculation uses PBE-PAW pseudopotentials for both atom species.
- **CP2K:** the input dataset performs an ab-initio molecular dynamics (AIMD) of liquid water using the Born-Oppenheimer approach, using Quickstep DFT. Specifically, the system is made of 256 water molecules (768 atoms, 2048 electrons) in a 19.7 cubic angstrom cell and molecular dynamics is run for 10 steps. Production quality settings for the basis sets (TZV2P) and the planewave cutoff (280 Ry) are chosen, and the Local Density Approximation (LDA) is used for the calculation of the Exchange-Correlation energy. The configurations were generated by classical equilibration, and the initial guess of the electronic density is made based on atomic orbitals.
- **Yambo:** the input dataset calculates QP corrections using the Yambo executable at the GW-PPA level. The system is constituted by a CH_4 (methane) molecule in a fcc cell, R=15 A. The system counts 4 valence bands. DFT calculations have been performed using pw.x of Quantum ESPRESSO and adopting norm-conserving pseudopotentials with a kinetic energy cutoff (for wavefunctions) of 90 Ry. The electronic structure data generated by pw.x have been processed by p2y executable for conversion to the Yambo internal format.
- **Siesta:** the input dataset performs a SCF cycle using the Siesta executable, for a system composed of several images of a large Si quantum dot saturated with H (34560 electrons, 100 Ry of real-space grid cutoff). 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.
- **Fleur:** the input dataset performs a SCF iteration for one k-point using *fleur_MPI* executable of a system constituted by a supercell 2 x 2 x 4 of MnGe (128 atoms). This system is a non-centrosymmetric B20 chiral magnet which has a property of breaking the inversion symmetry leading to various types of magnetic structures. Therefore the calculation includes non-collinear magnetism and spin-orbit coupling. It uses the PBE exchange-correlation functional.
- **BigDFT:** the input dataset performs a DFT calculation of a system composed of a boron fullerene B_{80}, specifically 80 boron atoms, 120 orbitals. The calculations use GGA-PBE exchange-correlation functional. BigDFT application implements a systematic real-space wavelet basis, this input dataset uses a uniform wavelet grid with a spacing of 0.4 bohr with an extent of 11.5 bohrs for the coarse grid and 2.9 bohrs for the fine grid.

2.2.2. Software stack

We compile and link all the MaX codes using the fastest scientific libraries released from the hardware vendors. In particular, we use the following acceleration libraries for our target platforms:

[5] https://gitlab.com/max-centre/benchmarks

- **Intel Xeon Skylake**: We leverage Intel MKL[6] to provide accelerated routines for BLAS, LAPACK, ScaLAPACK, and FFTW for Xeon CPUs.
- **ARM ThunderX2**: for this architecture we used the ARM Performance Library[7] which provides BLAS, LAPACK optimized routines, while we used the vanilla version of FFTW[8] and the Netlib-ScaLAPACK[9] to distribute among all MPI processes LAPACK computation.
- **AMD Rome**: for the AMD platform, we leverage on the optimized routines released by AMD. In detail for BLAS we used the AMD Optimized BLIS[10], for LAPACK we leverage the AMD libFLAME[11], instead, the ScaLAPACK AMD release an optimized version to compute distributed lapack[12], and to conclude, AMD release an optimized version of FFT[13] for computing the Discrete Fourier Transform.
- **IBM Power9**: for the Power9 architecture we leveraged on the IBM ESSL[14], but due to the incompatibility of some codes with this library we also used the OpenBLAS[15] routines which provide BLAS, LAPACK, and optimized routines for Power architectures. We use Netlib-ScaLAPACK to distribute the LAPACK workload among multiple MPI processes.

Most MaX codes also require complex I/O read and write operations, for this reason, they often need I/O libraries to optimize block storage access. Thus, we utilize the HDF5 and the NETCDF libraries when supported by the application.

To exclude different compiler behaviours among several architectures, we chose to use the same compiler for all our target platforms. We selected GCC 9.3 because it is able to generate very efficient binaries for all our microarchitectures. Moreover, we used OpenMPI 4.1.1 for all the benchmarks because vendor optimization on network communication runtimes does not impact the efficiency of the application in single node runs, leaving the analysis of the network communication when the application runs at scale.

In Table 2, we show the entire software stack used for each benchmark with the exact versioning of the libraries:

Codes	Marconi Intel Skylake	Marconi100 IBM Power9	Armida ThunderX2	DGX AMD Rome
QE 6.4.1	mkl-2018	netlib-lapack-3.9.1 netlib-scalapack-2.1.0 fftw-3.3.8	armpl-20.3.0 scalapack-2.1.0	amdblis-3.0 amdlibflame-3.0 amdscalapack-3.0 amdfftw-3.0
QE 6.7 MaX Release	mkl-2018	netlib-blas-3.8.0 netlib-lapack-3.9.0 scalapack-2.1.0 fftw-3.3.8	armpl-20.3.0	amdblis-3.0 amdlibflame-3.0 amdscalapack-3.0 amdfftw-3.0

[6] https://www.intel.com/content/www/us/en/developer/tools/oneapi/onemkl.html
[7] https://www.arm.com/products/development-tools/server-and-hpc/allinea-studio/performance-libraries
[8] https://www.fftw.org
[9] http://www.netlib.org/scalapack
[10] https://developer.amd.com/amd-aocl/blas-library
[11] https://developer.amd.com/amd-aocl/blas-library/#libflame
[12] https://developer.amd.com/amd-aocl/scalapack
[13] https://developer.amd.com/amd-aocl/fftw
[14] https://www.ibm.com/docs/en/essl/6.2
[15] https://www.openblas.net

Yambo 4.5.3	mkl-2018 netcdf-fortran-4.5.3 netcdf-c-4.8.1 hdf5-1.10.7	openblas-0.3.18 netlib-scalapack-2.1.0 fftw-3.3.9 netcdf-fortran-4.5.3 netcdf-c-4.8.1 hdf5-1.10.7	armpl-20.3.0 netlib-scalapack-2.1.0 fftw-3.3.9 netcdf-fortran-4.5.3 netcdf-c-4.8.1 hdf5-1.10.7	amdblis-3.0 amdlibflame-3.0 amdscalapack-3.0 amdfftw-3.0 netcdf-fortran-4.5.3 netcdf-c-4.8.1 hdf5-1.10.7
Yambo 5.0.4	mkl-2018 netcdf-fortran-4.5.3 netcdf-c-4.8.1 hdf5-1.10.7	openblas-0.3.18 netlib-scalapack-2.1.0 fftw-3.3.9 netcdf-fortran-4.5.3 netcdf-c-4.8.1 hdf5-1.10.7	armpl-20.3.0 netlib-scalapack-2.1.0 fftw-3.3.9 netcdf-fortran-4.5.3 netcdf-c-4.8.1 hdf5-1.10.7	amdblis-3.0 amdlibflame-3.0 amdscalapack-3.0 amdfftw-3.0 netcdf-fortran-4.5.3 netcdf-c-4.8.1 hdf5-1.10.7
CP2K 6.1	mkl-2018	openblas-0.3.17 netlib-scalapack-2.1.0 fftw-3.3.9	armpl-20.3.0 netlib-scalapack-2.1.0 fftw-3.3.9	amdblis-3.0 amdlibflame-3.0 amdscalapack-3.0 amdfftw-3.0
CP2K 8.1	mkl-2018	openblas-0.3.17 netlib-scalapack-2.1.0 fftw-3.3.9	armpl-20.3.0 netlib-scalapack-2.1.0 fftw-3.3.9	amdblis-3.0 amdlibflame-3.0 amdscalapack-3.0 amdfftw-3.0
Siesta 4.0.2	mkl-2019	netlib-lapack-3.9.0 essl-6.2.1 netlib-scalapack-2.1.0	armpl-20.3.0 netlib-scalapack-2.1.0	amdblis-3.0 amdlibflame-3.0 amdscalapack-3.0
Siesta MaX 1.3.1	mkl-2018 elsi-2.7.1	netlib-lapack-3.9.0 essl-6.2.1 netlib-scalapack-2.1.0 elsi-2.6.2	armpl-20.3.0 netlib-scalapack-2.1.0 elsi-2.7.1	amdblis-3.0 amdlibflame-3.0 amdscalapack-3.0 elsi-2.7.1
Fleur MaX-R3.1	openblas-0.3.18 netlib-scalapack-2.1.0 fftw-3.3.9 hdf5-1.1.0.7	openblas-0.3.18 netlib-scalapack-2.1.0 fftw-3.3.9 hdf5-1.1.0.7	armpl-20.3.0 netlib-scalapack-2.1.0 fftw-3.3.9 hdf5-1.1.0.7	amdblis-3.0 amdlibflame-3.0 amdscalapack-3.0 amdfftw-3.0 hdf5-1.1.0.7
Fleur MaX-R5.1	openblas-0.3.18 netlib-scalapack-2.1.0 fftw-3.3.9 hdf5-1.1.0.7	openblas-0.3.18 netlib-scalapack-2.1.0 fftw-3.3.9 hdf5-1.1.0.7	armpl-20.3.0 netlib-scalapack-2.1.0 fftw-3.3.9 hdf5-1.1.0.7	amdblis-3.0 amdlibflame-3.0 amdscalapack-3.0 amdfftw-3.0 hdf5-1.1.0.7
BigDFT 1.8.2	mkl-2018	openblas-0.3.18 netlib-scalapack-2.1.0	armpl-20.3.0 netlib-scalapack-2.1.0	amdblis-3.0 amdlibflame-3.0 amdscalapack-3.0
BigDFT 1.9.1	mkl-2018	openblas-0.3.18 netlib-scalapack-2.1.0	armpl-20.3.0 netlib-scalapack-2.1.0	amdblis-3.0 amdlibflame-3.0 amdscalapack-3.0

Table 2: the entire software stack used for each benchmark with the exact versioning of the libraries.

2.3. Profiling tools

We conducted the profiling analysis of the codes using two profiling tools, Linux Perf and COUNTDOWN.

2.3.1. Linux Perf

Linux Perf is an event-oriented observability tool, which we used to collect performance metrics from the microarchitecture of the system. It works across different platforms since it is part of the Linux kernel. Initially developed to query the performance counters subsystem in Linux, it has been enhanced to add tracing capabilities. Fig. 3 shows several HW and SW events made possible using it . For this analysis, though Linux Perf is capable of tracing several metrics from both software and hardware level, we only focused on HW performance metrics, sampling the relative HW performance counters (PMCs) available in the system. PMCs are CPU hardware registers that count hardware events such as instructions executed, CPU cycles, and memory operations.

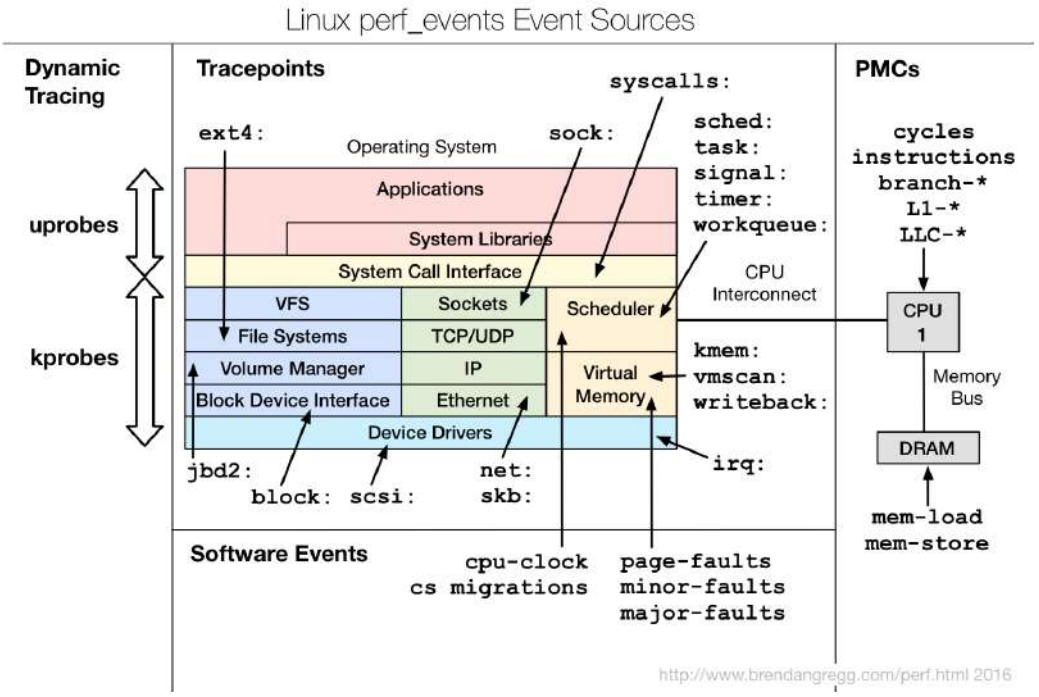

Fig. 3: A general overview of Linux Perf profiling capabilities.

Each platform has its specific PMCs, and this makes it difficult to calculate the same derived metrics and to compare them among different architectures.

2.3.2. COUNTDOWN

The second tool we used is an open-source tool developed by the University of Bologna and CINECA, called COUNTDOWN[16] [17]. This tool was initially developed to improve the energy efficiency of HPC applications, and now it implements a profiler capable of monitoring the power management subsystems of our target platforms. Here we have used COUNTDOWN as a profiler to extract the power metrics of the applications. The use of this tool has offered the chance of creating a point of contact between the MaX project and the REGALE project[18] in which COUNTDOWN is one of the principal tools used to create the first European software stack for power management at exascale.

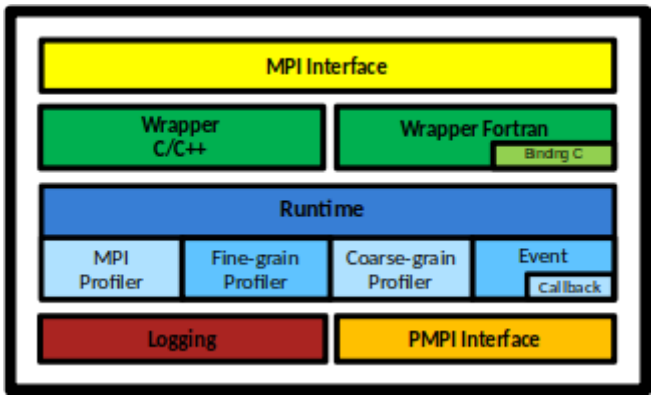

Fig. 4: The stack components of the COUNTDOWN tool.

COUNTDOWN can automatically instrument C/C++ and Fortran applications and uses a PMPI interface to intercept all MPI calls from the application to the MPI library. Every second, it samples the power controls of the HW subsystem, where it is able to profile the energy consumed by the application. It supports the Intel architecture through the interaction with Intel RAPL[19], the IBM Power9 with the interaction with the On-Chip Controller[20] (OCC), and with the ThunderX2 through the TX2MON[21] driver of the platform. Fig. 4 shows the main components of COUNTDOWN, in particular, we focus on the MPI and the PMPI interface used to instrument the application, and on the profilers used to monitor the PMC and the MPI communication. COUNTDOWN is an open source tool publicly available on Github[22].

[16] D. Cesarini, A. Bartolini, P. Bonfà, C. Cavazzoni, and L. Benini, "COUNTDOWN: A Run-Time Library for Performance-Neutral Energy Saving in MPI Applications," in IEEE Transactions on Computers, vol. 70, no. 5, pp. 682-695, 1 May 2021, doi: 10.1109/TC.2020.2995269.
[17] D. Cesarini, A. Bartolini, A. Borghesi, C. Cavazzoni, M. Luisier, and L. Benini, "Countdown Slack: A Run-Time Library to Reduce Energy Footprint in Large-Scale MPI Applications," in IEEE Transactions on Parallel and Distributed Systems, vol. 31, no. 11, pp. 2696-2709, 1 Nov. 2020, doi: 10.1109/TPDS.2020.3000418.
[18] https://regale-project.eu
[19] https://01.org/blogs/2014/running-average-power-limit-%E2%80%93-rapl
[20] https://on-demand.gputechconf.com/gtc/2015/presentation/S5699-Todd-Rosedahl-OpenPOWER.pdf
[21] https://github.com/Marvell-SPBU/tx2mon
[22] https://github.com/EEESlab/countdown

2.4. Platform targets and microarchitectures

In our analysis, we identify four HPC systems with different CPUs to cover common architectures used in HPC. So, we have been able to compare different performances, energy efficiency, and performance portability on the MaX codes, among different platforms. In the following paragraph, we present the microarchitecture of the target systems and our effort to understand their characteristics and to analyze how MaX codes run efficiently (or not) on these systems.

2.4.1. Marconi - Intel Xeon Skylake

The first HPC system we used in this analysis is Marconi hosted at CINECA, a part of the Eurofusion system[23]. Marconi nodes are double-socket servers based on Intel Xeon 8160 (Skylake) CPUs equipped with 24 cores at 2.1GHz (nominal frequency) and 192GB of DDR4.

Fig. 5 shows the microarchitecture of a Xeon Skylake core organized in the following subsystems: i) *Front end*, ii) *Execution Engine*, and iii) *Memory Subsystem*.

[23] https://www.hpc.cineca.it/news/eurofusion-marconi-selected-support-european-research-fusion

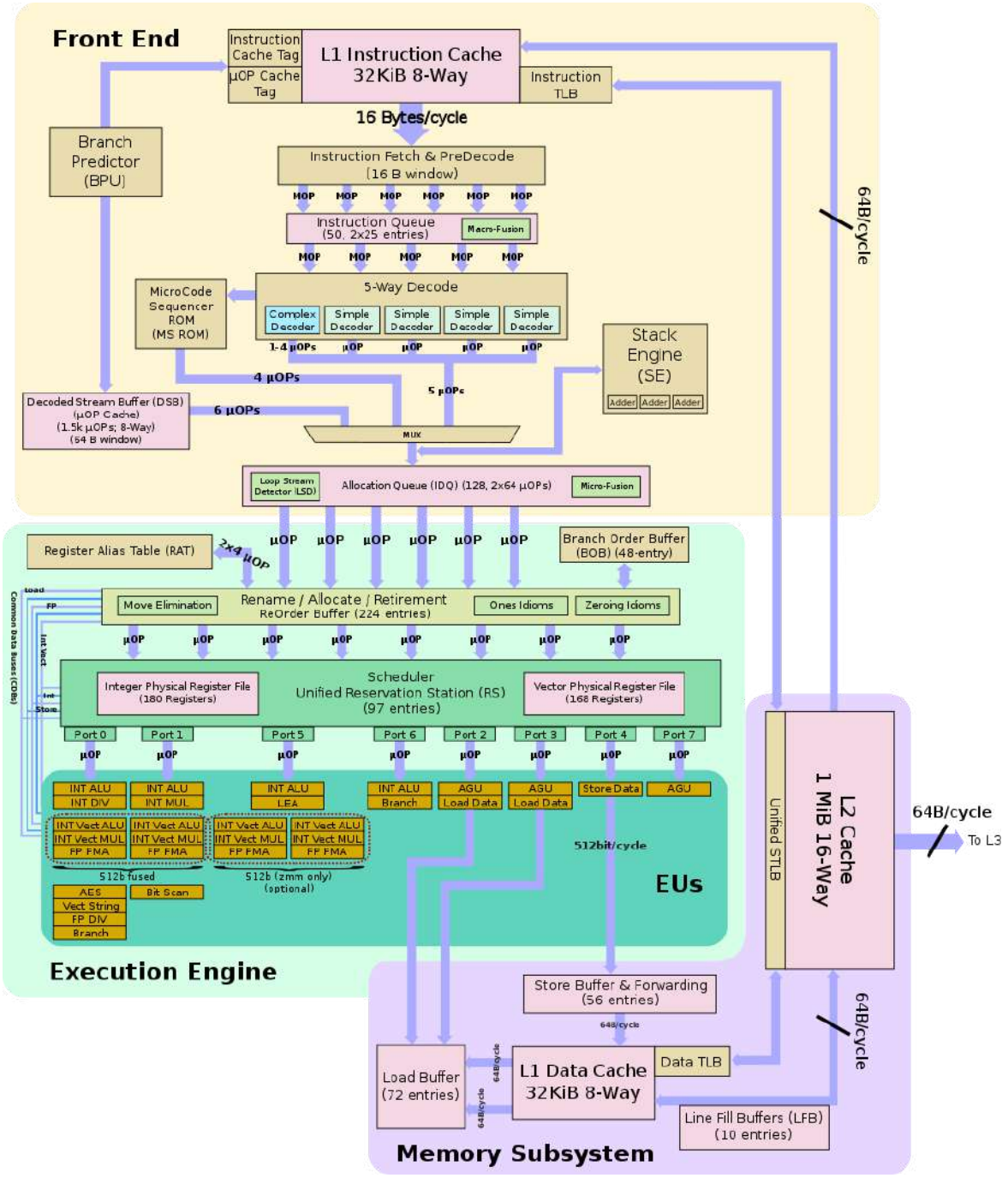

Fig. 5: Overview of the Intel Xeon Skylake microarchitecture[24].

While Fig. 5 reports several details of the microarchitecture for completeness, we want to focus only on a small subset of HW units relevant for our analysis.

First of all from the microarchitecture shown in Fig. 5, we extract the information about the IPC. The Front End group spots the 5-way decode unit, responsible for decoding x86 instructions, that allows the Skylake microarchitecture to decode up to 5 x86 instructions (or Macro Operation, MOP) per

[24] https://en.wikichip.org/wiki/intel/microarchitectures/skylake_(server)#Front-end

cycle into micro operations (μOP). This architecture can therefore deliver a theoretical IPC of 5. Due to various other more restricting points in the pipeline[25], we can practically reach only an IPC of 4.

Secondly we have considered the vectorization units. The Xeon Skylake microarchitecture is equipped with two AVX-512 execution units capable of delivering 32 FLOPs/cycle in double precision (64bit length). The FLOPs/cycle are calculated by multiplying the FLOPs capability of a AVX-512 execution units by two vectors, and to multiply by another 2 if we consider floating point multiply-accumulate operation (FMA):

$$[(512bit \div 64bit) \times 2\ vector\ units] \times 2\ FMA = 32\ \frac{FLOPs}{cycle} DP$$

In Fig. 5, considering the two execution units in the *Execution Engine* group, take into account that the AVX-512 execution units are linked only to the port 1, 2, and 5, which link the instruction *Scheduler* to the execution units. While port 5 can deliver a single operand of 512bit, port 1 and 2 can only deliver 256bit operands (probably to support AVX-256bit), so an AVX-512 operation requires both ports to execute (512bit fused), limiting the execution of two AVX-512 per cycle. The Xeon 8120 CPU is equipped with 24 cores at 2.1GHz (AVX-512 operating frequency), so a double-socket node can theoretically deliver up to 3.2 TFLOPs of peak performance.

While this CPU can deliver a high number of FLOPs, it has a limited performance in terms of memory bandwidth with respect to the other target platforms, since it is equipped with only 6 memory controllers with DDR4 at 2666 MT/s supporting a memory throughput up to 128 GB/s.

From this CPU we expect that MaX codes to run very efficiently even if limited to the memory throughput of the architecture.

PMCs and Metrics of Intel Xeon Skylake

We use a subset of PMCs available in the microarchitecture to collect the information we have used to calculate the derived metrics of our analysis. We report in the following the PMCs for Intel Xeon Skylake:

PMC	**Description**
INST_RETIRED	Number of instructions executed
CPU_CLK_UNHALTED.THREAD	Core clock cycles whenever the clock signal on the specific core is running (not halted)
FP_ARITH_INST_RETIRED.SCALAR_DOUBLE	Number of scalar double precision floating-point arithmetic instructions executed (multiply by 1 to get FLOPs). FMA instructions are counted twice.
FP_ARITH_INST_RETIRED.128B_PACKED_DOUBLE	Number of scalar 128-bit packed double precision floating-point arithmetic instructions executed (multiply by 2 to get FLOPs). FMA instructions are counted twice

[25] https://en.wikichip.org/wiki/intel/microarchitectures/skylake_(server)#Front-end

FP_ARITH_INST_RETIRED.256B_PACKED_DOUBLE	Number of scalar 256-bit packed double precision floating-point arithmetic instructions executed (multiply by 4 to get FLOPs). FMA instructions are counted twice
FP_ARITH_INST_RETIRED.512B_PACKED_DOUBLE	Number of scalar 512-bit packed double precision floating-point arithmetic instructions executed (multiply by 8 to get FLOPs). FMA instructions are counted twice
UOPS_ISSUED.ANY	Number of µOPs issued by the Resource Allocation Table (RAT) to the Reservation Station (RS). This PMC is used to calculate the TMAM
UOPS_RETIRED.RETIRE_SLOTS	Number of retirement slots used. This PMC is used to calculate the TMAM
IDQ_UOPS_NOT_DELIVERED.CORE	Number of µOPs not delivered to the RAT per thread. This PMC is used to calculate the TMAM
INT_MISC.RECOVERY_CYCLES	Number of the core cycles the allocator was stalled due to recovery from earlier clear events for any thread running on the physical core (e.g. misprediction). This PMC is used to calculate the TMAM
MSR_PKG_ENERGY_STATUS	Measure the actual energy usage by the package. It is necessary to multiply with the value reported in the register *MSR_RAPL_POWER_UNIT* within bits 12:8 to convert this value into micro joules

Table 3: PMCs for Intel Xeon Skylake.

While it is not possible to monitor all the PMCs at the same time, we run the same application, with the same dataset input, on the same compute node multiple times until we collect all the PMC values.

After that, we were able to calculate the following derived metrics presented in this analysis:

Derived Metrics	**Formula**
AVG IPC	INST_RETIRED / CPU_CLK_UNHALTED.THREAD
Double Precision FLOPs	(FP_ARITH_INST_RETIRED.SCALAR_DOUBLE + (FP_ARITH_INST_RETIRED.128B_PACKED_DOUBLE x 2) + (FP_ARITH_INST_RETIRED.256B_PACKED_DOUBLE x 4) + (FP_ARITH_INST_RETIRED.128B_PACKED_DOUBLE x 8)) / Execution time
Vectorization ratio	(FP_ARITH_INST_RETIRED.128B_PACKED_DOUBLE + FP_ARITH_INST_RETIRED.256B_PACKED_DOUBLE + FP_ARITH_INST_RETIRED.128B_PACKED_DOUBLE) / (FP_ARITH_INST_RETIRED.SCALAR_DOUBLE +

	FP_ARITH_INST_RETIRED.128B_PACKED_DOUBLE + FP_ARITH_INST_RETIRED.256B_PACKED_DOUBLE + FP_ARITH_INST_RETIRED.128B_PACKED_DOUBLE)
TMAM Retiring	UOPS_RETIRED.RETIRE_SLOTS / (4 * CPU_CLK_UNHALTED.THREAD)
TMAM Front end	IDQ_UOPS_NOT_DELIVERED.CORE / (4 * CPU_CLK_UNHALTED.THREAD)
TMAM Bad Speculation	(UOPS_ISSUED.ANY - UOPS_RETIRED.RETIRE_SLOTS + 4 * INT_MISC.RECOVERY_CYCLES) / (4 * CPU_CLK_UNHALTED.THREAD)
TMAM Back end	1 - (Retiring + Front End + Bad Speculation)
Package energy [J]	MSR_PKG_ENERGY_STATUS * MSR_RAPL_POWER_UNIT[12:8] (see Chapter 4.10)
AVG package power [W]	Package energy / Execution time
Energy Efficiency [FLOPs/W]	Double Precision FLOPs / AVG package power

Table 4: derived metrics (Intel Xeon Skylake).

2.4.2. Marconi100 - IBM Power9

The second architecture we targeted was a PowerPC, in particular we focused on the IBM Power9, the latest PowerPC of IBM. For this analysis we used the Marconi100 system in CINECA.

IBM Power9 is an IBM processor designed to support GPUs, we also used in our analysis to explore how MaX codes run on it. The microarchitecture of Power9 implements non-trivial design choices that set it apart from more general CPUs. First of all, each core implements a huge SMT technology which is capable of fetching up to 8 instructions per cycle and of decoding up to 6 instructions per cycle to maximize the instruction level parallelism. The CPU is equipped with 16 cores that can run up to 64 HW threads at the same time. Fig. 6 shows the architecture of a single core and all the HW units.

From Fig. 6, we see that a single core of a Power9 processor can dispatch up to 6 instructions per cycle, making this CPU a superscalar machine capable of delivering tens of billions of instructions per second.

The Power9 is equipped with two vector units at 128bit, each of which (represented in the execution unit called PM in Figure 6) can execute up to 2 FMA operations per cycle delivering up to 8 FLOPs/cycle for each core. This means that a 16-core CPU running at 3.8GHz (nominal frequency) can only deliver up to 486.4 GFLOPs per socket, for a total of 972.8 GFLOPs for a double-socket node. Thus, we can see that a single Power9 node is more than three times slower in terms of FLOPs than a Skylake node, but it is capable of executing 50% more of the instruction per cycle.

$$[(128bit \div 64bit) \times 2\ vector\ units] \times 2\ FMA = 8\ \frac{FLOPs}{cycle}\ DP$$

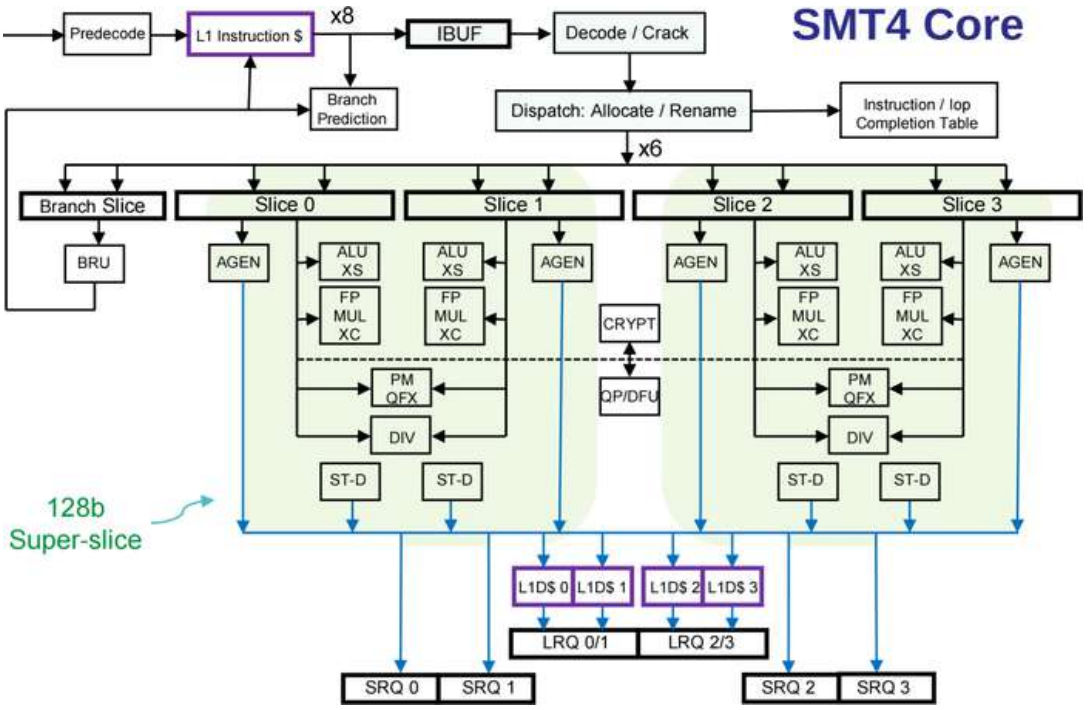

Fig. 6: Overview of the IBM Power9 microarchitecture[26].

While this CPU is very limited in terms of FLOPs performance, it is equipped with 8 memory controllers with DDR4 at 2666 MT/s, which can deliver up to 160GB/s. Moreover, IBM affirms that each Power9 is capable of 7 TB/s on-chip bandwidth; this such amount of performance is probably allowed by the 128bit cache line that Power9 microarchitecture implements, instead of the common 64bit cache line of all other target architectures. This feature makes this CPU very efficient for moving data inside of the chip.

So, Power9 systems are high-throughput systems designed for massive data moving and maintained at the same time as a high-execution unit to benefit high-throughput workload. We do not expect high performance from MaX code run on this system, since they are mainly bound to FLOPs performance, but exploring the application behaviours on this type of superscalar architecture to analyze the efficiency of the MaX codes is yet of interest.

[26] https://en.wikichip.org/wiki/ibm/microarchitectures/power9

PMCs and Metrics of IBM Power9

We report the PMCs that we used to calculate the microarchitectural metrics of our analysis:

PMC	**Description**
PM_RUN_INST_CMPL	Number of instructions executed
PM_RUN_CYC	Count core clock cycles
PM_SP_FLOP_CMPL	Floating point instruction completed in single precision
PM_DP_QP_FLOP_CMPL	Floating point instruction completed in double and quad precision
PM_SCALAR_FLOP_CMPL	Scalar floating point operation completed in single and double precision
PM_VECTOR_FLOP_CMPL	Vector floating point operation completed both in single and double precision
PM_FMA_CMPL	Two flops operations completed (fmadd, fnmadd, fmsub, fnmsub) only for scalar instructions both single and double precision
PM_NON_FMA_FLOP_CMPL	Non FMA instruction completed both single and double precision
PM_MATH_FLOP_CMPL	Arithmetic instruction completed both single and double precision
PM_MBA[X]_READ_BYTES / PM_MBAX[X]_WRITE_BYTES	Number of read and write events of the integrated memory controller to move a cache line from/to the DDR memory. Each memory controller has its own READ and WRITE counter (*[X]* range from 0 to 8)

Table 5: PMCs for IBM Power9.

We report in the following the derived metrics used in this analysis for Power9 architecture:

Derived Metrics	**Formula**
AVG IPC	PM_RUN_INST_CMPL / PM_RUN_CYC
Double precision instruction ratio (DP inst ratio [%])	PM_DP_QP_FLOP_CMPL / (PM_DP_QP_FLOP_CMPL + PM_SP_FLOP_CMPL)
Scalar instruction ratio (SC inst ratio [%])	PM_SCALAR_FLOP_CMPL / (PM_SCALAR_FLOP_CMPL + PM_VECTOR_FLOP_CMPL)
Vectorization ratio (VC inst ratio [%])	1 - SC inst ratio
FMA instruction ratio	PM_FMA_CMPL / (PM_FMA_CMPL + PM_NON_FMA_FLOP_CMPL)

(FMA ratio [%])	
Non-FMA instruction ratio (Non-FMA ratio [%])	PM_NON_FMA_FLOP_CMPL / (PM_FMA_CMPL + PM_NON_FMA_FLOP_CMPL) or (1 - FMA ratio)
Double Precision FLOPs	((((DP inst ratio * PM_MATH_FLOP_CMPL) * SC inst ratio) + (DP inst ratio * PM_MATH_FLOP_CMPL) * VC inst ratio * 2) * Non-FMA ratio) + (((DP inst ratio * PM_MATH_FLOP_CMPL) * SC inst ratio) + (DP inst ratio * PM_MATH_FLOP_CMPL) * VC inst ratio * 2) * FMA ratio * 2)) / Execution Time
Memory Bandwidth [Byte/s] (for Roofline)	(SUM(PM_MBA[X]_READ_BYTES, PM_MBA[X]_WRITE_BYTES) * 64) / Execution time
Package energy [J]	From the on-chip controller, see Chapter 4.11
AVG package power [W]	Package energy / Execution time
Energy Efficiency [FLOPs/W]	Double Precision FLOPs / AVG package power

Table 6: Derived metrics (IBM Power9).

2.4.3. Armida - Cavium ThunderX2

In our target system we also include an ARM-based architecture for HPC CPUs. With the MaX partner E4, we identified as suitable for the task the ThunderX2 architecture on which the Armida system hosted at E4 is based. The ThunderX2 processors are at the base of the Astra System from Sandia which was the world's first petascale ARM supercomputer. ThunderX2 systems are based on Cavium Vulcan technology shown in Fig. 7.

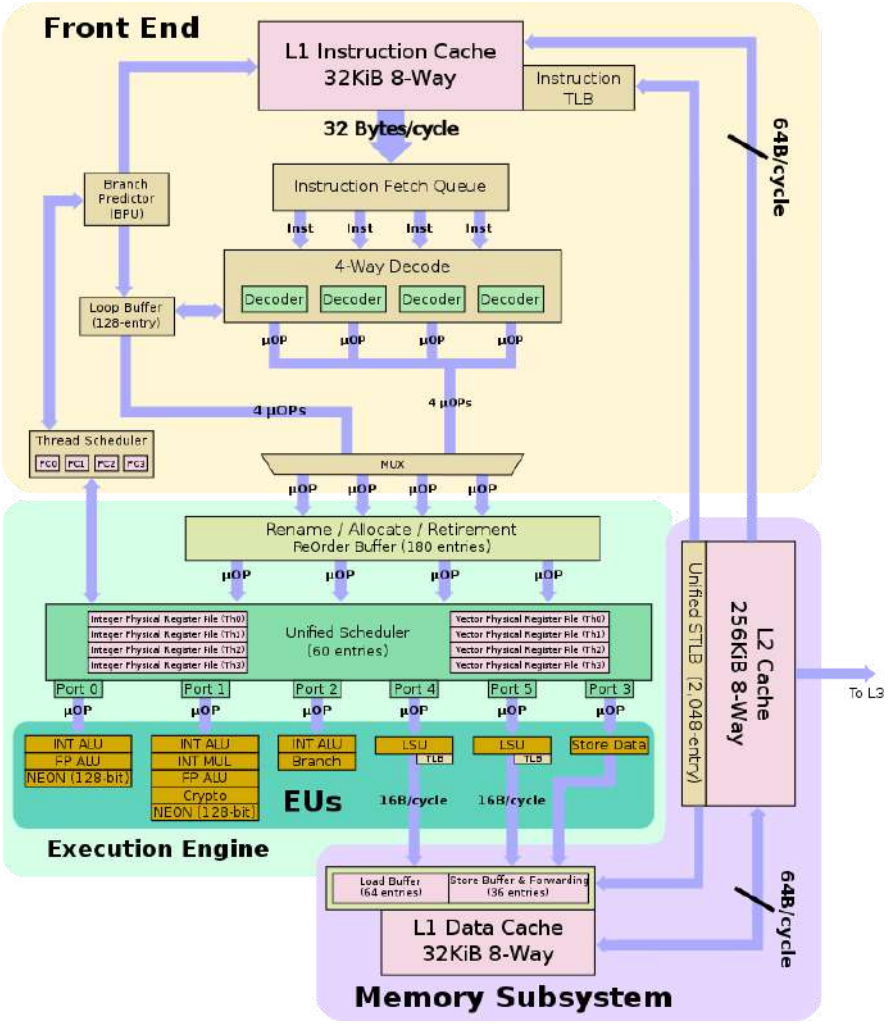

Fig. 7: Overview of the Vulcan microarchitecture equipped on the ThunderX2[27].

This microarchitecture is quite simple with respect to the Intel Skylake one; this allows the designers of the ThunderX2 supporting a high number of cores per socket, up to 32 single-thread cores.

Vulcan architecture can deliver up to 4 instructions per cycle, similar to Intel Skylake architecture, but it implements only two vector units at 128bit. This also means that it can deliver up to 8 FLOPs/cycle when executing FMA operations in a similar way to Power9. With an operating frequency up to 2.5GHz, a compute node equipped with a double socket can deliver up to 1.28 TFLOPs.

$$[(128bit \div 64bit) \times 2\ vector\ units] \times 2\ FMA = 8\ \frac{FLOPs}{cycle} DP$$

The ThunderX2 system supports 8 memory controllers of DDR4 at 2666 MT/s which deliver up to 160 GB/s of aggregate bandwidth in the same way as Power9 does.

From this CPU, we expect that the MaX codes will be limited from the execution units since the memory bandwidth should be more than enough to feed the back end of the core.

[27] https://en.wikichip.org/wiki/cavium/microarchitectures/vulcan

PMCs and Metrics of Armida ThunderX2

We report all the PMCs and metrics utilized in the Armida system to collect information about the microarchitecture efficiency of the MaX codes.

PMC	**Description**
INST_RETIRED	Number of instructions executed
CPU_CYCLES	Count core clock cycles
VFP_SPEC	Scalar floating point instruction executed
ASE_SPEC	Vector floating point instruction executed
STALL_FRONTEND	Cycle on which no instructions issued because there were no instructions ready to issue
STALL_BACKEND	Cycle on which no instructions issued due to back-end resources being unavailable
INST_SPEC	Instructions speculatively executed

Table 7: PMCs for the Armida ThunderX2.

In the following we present how we calculated the derived metrics for our analysis.

Derived Metrics	**Formula**
AVG IPC	INST_RETIRED / CPU_CYCLES
Double Precision FLOPs	(VFP_SPEC + (ASE_SPEC * 2)) / Execution Time
TMAM Front end	STALL_FRONTEND / CPU_CYCLES
TMAM Back end	STALL_BACKEND / CPU_CYCLES
TMAM Bad Speculation	((INST_SPEC - INST_RETIRED) / AVG IPC) / CPU_CYCLES
TMAM Retiring	1 - (Front end + Back end + Bad Speculation)
Package energy [J]	From the SoC management controller, see Chapter 2.5.3
AVG package power [W]	Package energy / Execution time
Energy Efficiency [FLOPs/W]	Double Precision FLOPs / AVG package power

Table 8: Derived metrics (Armida ThunderX2).

2.4.4. DGX - AMD Rome

Our last target system is an Nvidia DGX, based on an AMD 7742 Epyc processor. Fig. 8 shows the microarchitecture of the Zen2 microarchitecture on which AMD EPYC 7742 is based.

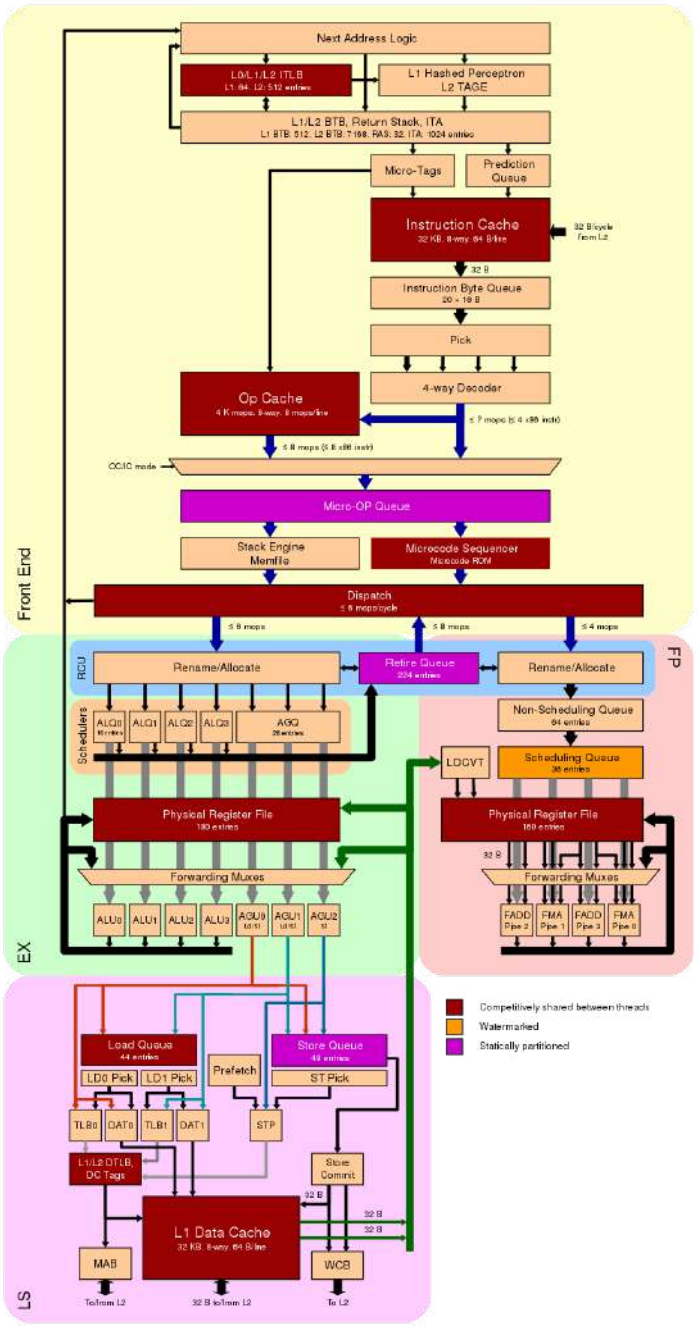

Fig. 8: Overview of the AMD Zen2 microarchitecture[28].

[28] https://en.wikichip.org/wiki/amd/microarchitectures/zen_2

This is an architecture as complex as the Intel Skylake one, capable of delivering up to 4 instructions per cycle. This is visible in the *4-way Decoder* in the *Front End* part of the CPU shown in Fig. 8. The IPC can go over 4 if the code reutilizes chunks of instructions previously decoded and temporarily stored in an operation cache delivering up to 8 x86 instructions.

The floating point units of AMD Zen2 architecture is quite complex: basically it uses a coprocessor architectural model where all floating point operations are executed. It can execute up to 2 execution units at the same time, capable of FMA operations; this means that each core can deliver up to 16 FLOPs/cycle in double precision. With an operating frequency of 2.25 GHz and 64 cores for each socket, the AMD 7742 Epyc processor can deliver up to 2.3 TFLOPs for a total of 4.6 TFLOPs of a double socket node.

$$[(256bit \div 64bit) \times 2\ vector\ units] \times 2\ FMA = 16\ \frac{FLOPs}{cycle} DP$$

The AMD 7742 supports 8 memory controllers of DDR4 at 3200 MT/s and it can deliver up to 190 GB/s of aggregate bandwidth making this CPU the most powerful on our target architectures in terms of IPC, FLOPs, and memory bandwidth.

PMCs and Metrics of DGX AMD Rome

As for the other architecture, we report the PMCs used for this analysis:

PMC	**Description**
INST_RETIRED	Number of instructions executed
CPU_CYCLES	Count core clock cycles
RETIRED_SSE_AVX_FLOPS	The number of retired SSE/AVX FLOPs

Table 9: PMCs for DGX AMD Rome.

Due an old kernel on which our DGX are based, we are limited to query the average IPC and the FLOPs delivered by the applications calculated as follows:

Derived Metrics	**Formula**
AVG IPC	INST_RETIRED / CPU_CYCLES
Double Precision FLOPs	RETIRED_SSE_AVX_FLOPS / Execution Time

Table 10: derived metrics (DGX AMD Rome).

2.5. Power management subsystems

We conduct our analysis not only focusing on the performance of the code and the efficiency of the microarchitecture of the CPU, but taking into account the energy efficiency of the code. Due to the fact that energy consumption is not a standard architectural metric to monitor, we present in the

following subchapters the mechanisms implemented in our platform to collect information on the energy consumption of the systems.

2.5.1. Intel Xeon Skylake

Intel architecture implements an off-core mechanism called Running Average Power Limit (RAPL) that provides a set of counters providing for energy and power consumption information.

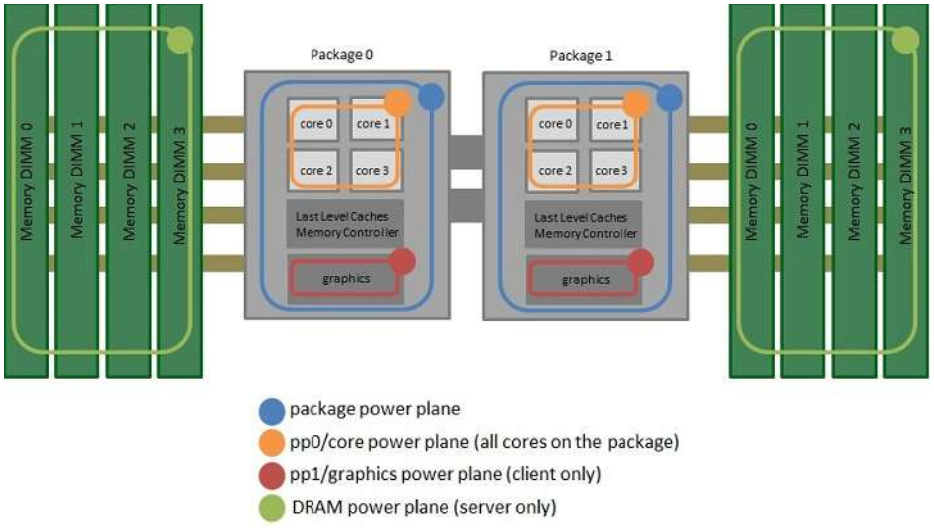

Fig. 9: Overview of the Intel RAPL architecture for a dual-socket system[29].

RAPL is organized in four power domains that handle different components on the system. Fig. 9 shows a global overview of all power domains in a double socket system. The power domains are:

- **Package domain**: this power domain contains information on the energy consumption of all components of the package, both cores and uncore energy consumption.
- **DRAM domain**: this domain is focused on monitoring the energy consumption of DRAM memory. This domain is present only in the server version of the Intel architecture (Xeon).
- **PP0/core power plane**: this domain monitors the energy consumption of the cores.
- **PP1/graphic power plane**: this domain is present only in the client version of Intel architectures and it is responsible for monitoring the energy consumption of the graphic HW unit of Intel processors.

The power domains are sampled every millisecond from the RAPL unit and the energy number is accumulated in a performance counter expressed in microjoules.

RAPL is not limited to energy monitoring but it can also power capping the CPU by setting a time window that is used to respect a user-defined power limit. RAPL monitors the average consumption in an interval specified in the time window and it reduces the operating frequency of the CPU if the power cap is not respected.

Unfortunately, in Intel architectures it is not possible to collect information on the energy consumption of the entire node, limiting our analysis to the package.

[29] https://blog.chih.me/read-cpu-power-with-RAPL.html

2.5.2. IBM Power9

The Power8/9 processor implements a hardware mechanism based on a dedicated microcontroller (IBM PowerPC 405) that handles power, performance, and thermal of the entire system. This HW mechanism is called On-Chip Controller (OCC) and it is part of the Power8/9 processor. The OCC provides access to detailed chip temperature, power, and utilization data, as well as complete control of CPU frequency, voltage, and memory bandwidth. Differently from RAPL, OCC can interact with components outside of the chip package through the use of SPI and I2C bus. OCC can also monitor the energy consumption of the entire node thanks to the interaction with the Board Management Control (BMC). It is also capable of monitoring the energy consumption of GPUs if they are present in the node. It exposes information on energy, power and temperature out of band through the network of the BMC, and in band through a specific device driver mounted in the OS kernel. The information reported from the OCC is very fine-grained; it is possible to monitor the power consumption of each core, of the entire package, of each DRAM DIMM, of each GPU, and of the entire system node. OCC samples the power sensors about every 250 microseconds[30] making the monitoring of the energy very accurate. In Fig. 10, the main components of the OCC and the interconnection with the other components on- and off-package are shown.

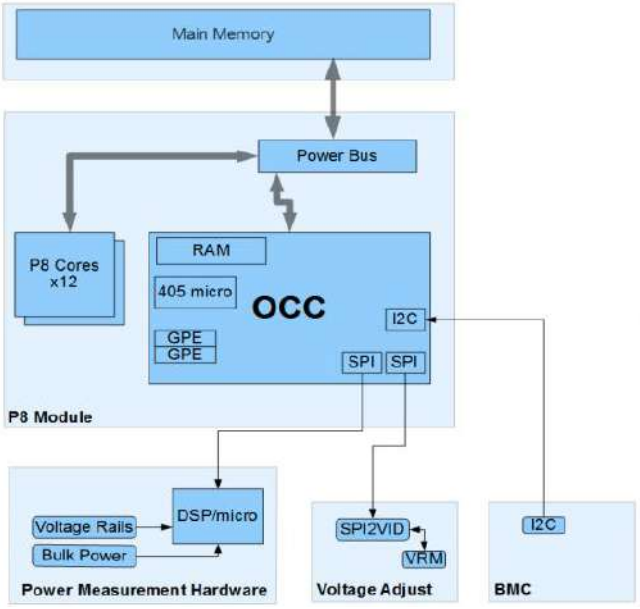

Fig. 10: Overview of On-Chip Controller of Power8/9 processor[31].

We use the information coming from OCC to profile the energy consumption of the MaX codes on the Power9 architecture. We only use package and DRAM information to be fairly in comparison with other target architectures.

30 https://cdn.openpowerfoundation.org/wp-content/uploads/2016/11/Todd-Rosedahl-Measuring-and-Managing-Power-Consumption.pdf

31 https://on-demand.gputechconf.com/gtc/2015/presentation/S5699-Todd-Rosedahl-OpenPOWER.pdf

2.5.3. Cavium ThunderX2

Similar to the Power9 OCC, ThunderX2 system implements a SoC management controller (MC), which takes care of power, temperature, and frequency of the CPU. Cavium released an open-source kernel module for Linux called *tx2mon*[32] capable of decoding and displaying the operating region of the MC from the Kernel level. The driver reports the current temperature from sensors, operating frequencies, and power measurements that are logged by the MC to host memory. *tx2mon* reports the following power domains:

- **Core**: voltage and power consumed by all cores on the SoC.
- **SRAM**: voltage and power consumed by internal SRAM on the SoC.
- **Internal memory**: voltage and power consumed by the LLC ring on the SoC.
- **SoC**: voltage and power consumed by miscellaneous SoC blocks.

tx2mon reports on the sensors monitored by the MC. Some of the on-chip components (like the various IO blocks) do not have sensors and therefore voltage and power measurements of these blocks are not provided by *tx2mon.* This differs from the other power controllers as it only reports the current power consumption and energy consumption. As the power consumption is sampled every second, we are able to estimate only partial energy consumption and our energy analysis is not too accurate. On the contrary, we cannot increase the sampling of our energy monitoring tool to avoid high overheads. For this reason, the energy consumption on the ThunderX2 is only an estimate of the real energy consumption of the CPU.

2.5.4. AMD Rome

The AMD Epyc processor implements a RAPL interface similar to the Intel Xeon architecture's one. AMD released an AMD energy driver to map the RAPL HW interface in the sysfs as the Intel RAPL does. This driver has been released from the Kernel Linux 5.8[33] which is a quite recent version of the Linux Kernel. Unfortunately, our DGX systems use a Linux Kernel older than 5, so in our analysis we are not able to monitor the energy consumption of this system.

We report in Table 11 the main characteristics of our target architecture, along with thermal design power (TDP) of each CPU and the energy efficiency expressed in GFLOPs/W.

CPU	Cores	SMT	Operating Frequency (GHz)	Peak IPC	Peak FLOPs (GFLOPs)	Peak Memory bandwidth (GB/s)	TDP (W)	Peak Energy Efficiency (GFLOPs/W)
Intel Xeon 8160	24	OFF	2.1 (AVX-512)	4 (6)	1612	120	150	10.75
IBM Power 9	16	x4	3.8	6	486	160	250	1.94
Cavium ThunderX2	32	OFF	2.5	4	640	160	200	3.20

[32] https://github.com/jchandra-cavm/tx2mon
[33] https://www.phoronix.com/scan.php?page=news_item&px=AMD-Energy-Driver-Linux

AMD 7742 Epyc	64	OFF	2.25 (AVX-256)	4 (8)	2662	190	225	11.82

Table 11: main characteristics of our target architecture, thermal design power (TDP) of each CPU and the energy efficiency expressed in GFLOPs/W.

2.6. Experimental Results

In this chapter we show the experimental results using the methodologies above presented to analyze the efficiency of the MaX codes. For each code we compare a version older than the project with the latest one released during the project. We put under the spotlight the overall performance, the microarchitecture efficiency, and the energy efficiency, using different computing platforms at the base of most of today's supercomputers. This analysis is not aimed at showing all the progress done on the codes, but rather at presentinging a general overview of the efficiency of the codes out of the box.

2.6.1. Quantum ESPRESSO

The first MaX code that we have analyzed is Quantum ESPRESSO (QE). In Fig. 11, we show all the exploration results we collected from QE on different platforms. We order the bars from left to right in performance FLOPs order, where DGX is the fastest node and the Marconi100 is the slowest one.

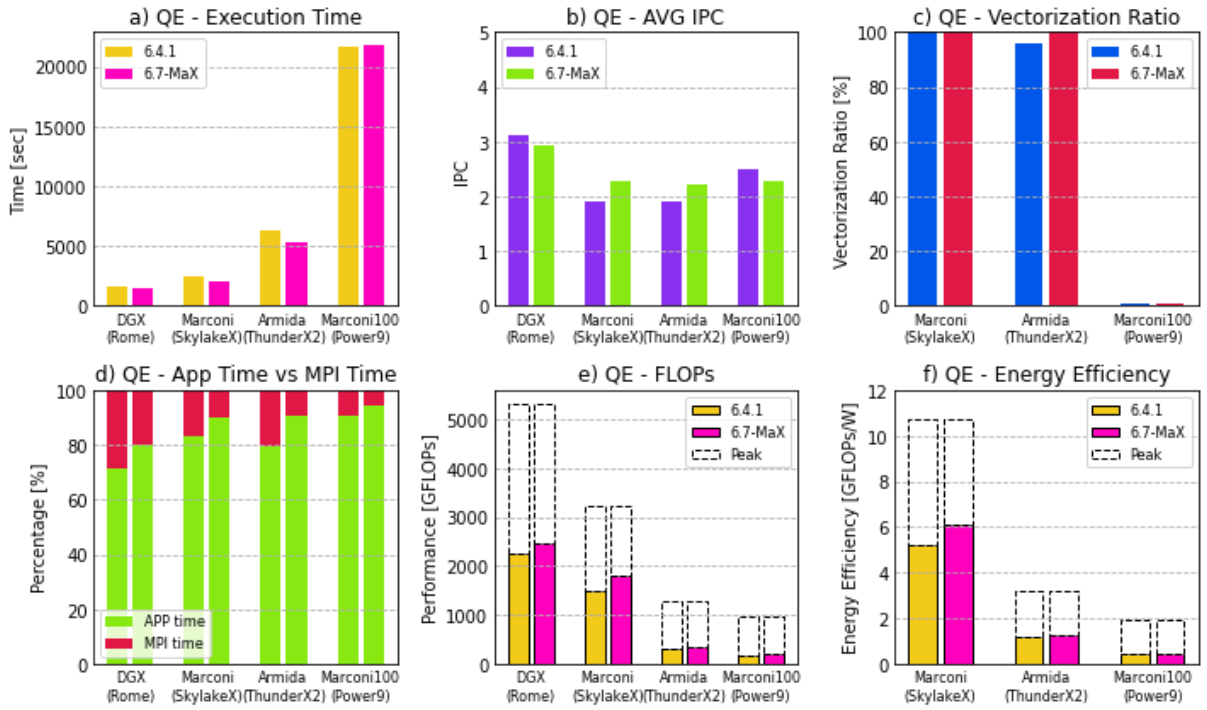

Fig. 11: Experimental results of Quantum ESPRESSO 6.4.1 vs 6.7-MaX.

We report the experimental results for both 6.4.1 (pre-MaX project) and 6.7-MaX (in-MaX project) release of QE. First of all in Fig. 11.a, we can note that the execution time is strictly related to the

FLOPs performance of the CPU also shown in Fig. 11.e and the code is able to extract more than 50% of the peak FLOPs performance, which is a very high efficiency. This proves that QE is a very efficient code in terms of FLOPs performance; this is also supported from the vectorization level of the code reported in Fig. 11.c. The vectorization ratio for Marconi and Armida reports 100%, that is clearly impossible, but the PMCs of Intel Xeon architectures are not able to distinguish between FMA and non-FMA instructions, considering the FMA operations as two instructions executed going beyond 100% of vectorization. From Fig. 11.c we only want to point out a quality of the vectorization ratio of the code without identifying an exact number.

Although the code shows a high vectorization level, the IPC for all the target platforms is quite high with an average level greater than 50% of the theoretical peak. This means a good performance balance between the front-end and the back-end of the core, which are able to keep the vector execution units busy without undersupplying. The microarchitecture of Intel Xeon Skylake is equipped with two vector units, this limits the IPC at 2 as the code is bounded by the vector units, but this is common for all of our target platforms. This also proves that the bottleneck of QE on all target architectures is the vector execution units which also limit the number of instructions executed per cycle. Consequently, the energy efficiency of QE shown in Fig. 11.f is very high for a limited power consumption of the chip (150W) and a very dense vectorization of the code.

We also present in Fig. 11.d the ratio between the application time and the MPI time, which shows how much time the application spent in the MPI library for communications by reporting the sum of the seconds for each core spent in the application code and in the MPI library. We can see from this plot a limited communication because the run is limited to a single node. The DGX reports the highest MPI time due to the fact that it also employs the highest number of MPI tasks with respect to the other platforms (128 MPI tasks).

Before we conclude, we want to show the only outlayer of our analysis, QE on the Marconi100 platform. For this benchmark the execution time increases exponentially with respect to the performance of the platform. This is an unexpected behaviour, but, as we see in Fig. 11.c, the vectorization level is almost zero and this causes the drop of performance. We think that this is caused by wrong vectorization of QE and scientific libraries. We didn't dig too deep on this issue, however we report here to underline how important the vectorization is and what impact this can have on the overall microarchitecture efficiency of the code.

We conclude that QE released within the MaX project shows a general improvement in all performance metrics with respect to the previous version, in particular in the microarchitecture efficiency on all platforms.

2.6.2. Fleur

The second benchmark in our analysis is Fleur. We ran it using the 3.1 version (pre-MaX project) and the latest stable version 5.1 (in-MaX project). Differently from QE, we show that Fleur is a memory bound code, as it is possible to understand by analyzing the time to solution in Fig. 12.a. We can see that the time-to-solution on Marconi is similar to Armida and Marconi100, which is the opposite of what happens to QE. It is interesting to note from Fig. 12.f that Marconi100 is able to reach around 50% of the FLOPs performance capability, a quite high figure comparable with the FLOPs of Marconi that can deliver more than 3x of peak FLOPs performance. We can exclude a vectorization issue

because, as reported in Fig. 12.c, it is still very high ≈50% for all our target platforms. So, this is clearly a data movement limitation: in fact, while Marconi100 has a lower performance in terms of FLOPs, it has a high data movement capability (memory and on-chip bandwidth) with respect to Marconi. Armida suffers from poor performance with respect to the other system, both in terms of time to solution and IPC, and we are not able to explain this effect by only reporting this data therefore in the Chapter in which we show experimental results on the TMAM analysis where we also provide an explanation.

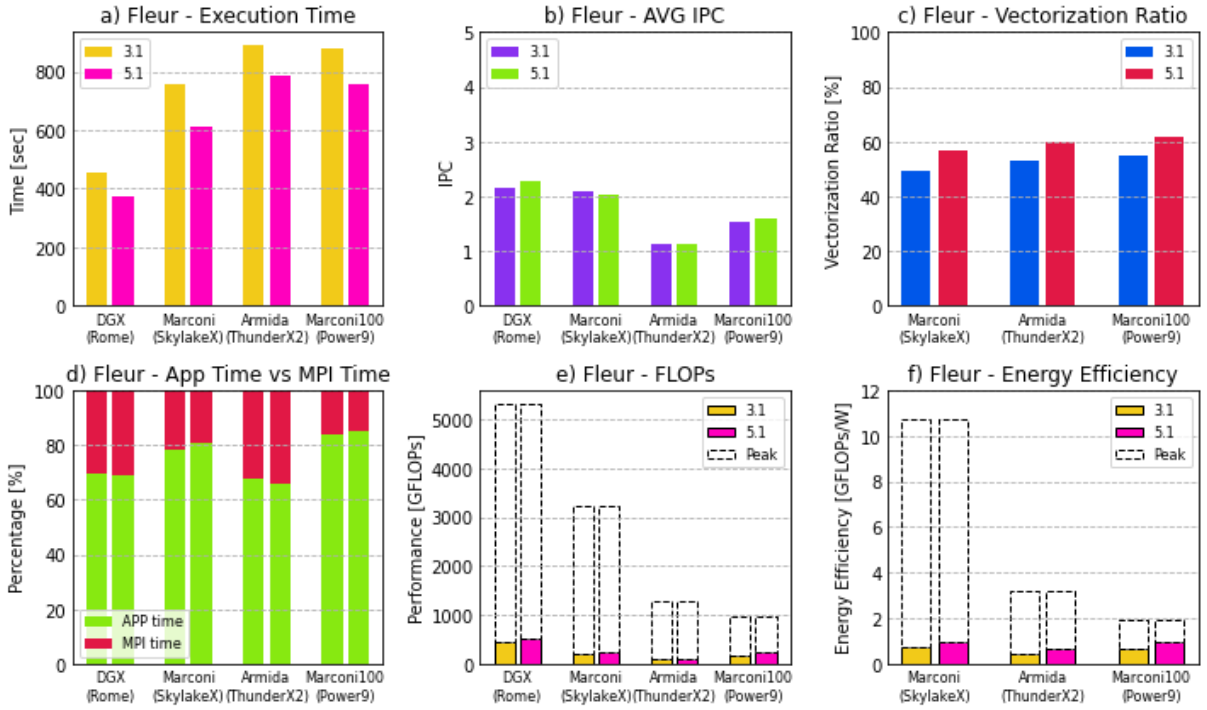

Fig. 12: Experimental results of Fleur 3.1 vs 5.1.

From the energy efficiency point of view, it is interesting to see that the lower energy efficient platform (Marconi100) with a peak efficiency of 1.94 GFLOPs/W is able to deliver the best energy efficient performance thanks to its high performance data movement capabilities.

The work done on Fleur in the Max project has also brought a brilliant improvement in the time to solution which is on average around 15% for all target platforms. The performance portability is quite high too, as all the platforms show a similar vectorization and IPC level.

2.6.3. CP2K

At first glance, CP2K's behaviour is similar to Fleur's in terms of memory boundness. E.g., the time to solution and the IPC shown in Fig. 13.a/b have the same trends of Fleur's.

We also see that CP2K is a very high vectorized code as shown in Fig. 138.c, and this ought to make a very efficient code, but looking at Fig. 13.d, CP2K presents a very huge time spent in MPI communication. CP2K shows on average >40% of the time spent in the MPI library and such communication transforms CP2K in a memory bound code limiting the overall FLOPs performance.

This is a very important categorization that differentiates CP2K from Fleur. While Fleur is a memory bound code CP2K looks like a network bound code. CP2K is able to deliver high FLOPs performance when the code of the application runs but this is limited due to the high amount of time spent in the MPI library to exchange data and for process synchronization purposes. We can also see that the main bottlenecks of CP2K are the communication model and the code itself. Consequently, the FLOPs performance and the energy efficiency are limited from the MPI time that does not contribute to reach the best performance in terms of FLOPs and energy efficiency. This analysis also shows that the performance of CP2K is more constrained to the communication boundness of the application, and this limits the scalability of code.

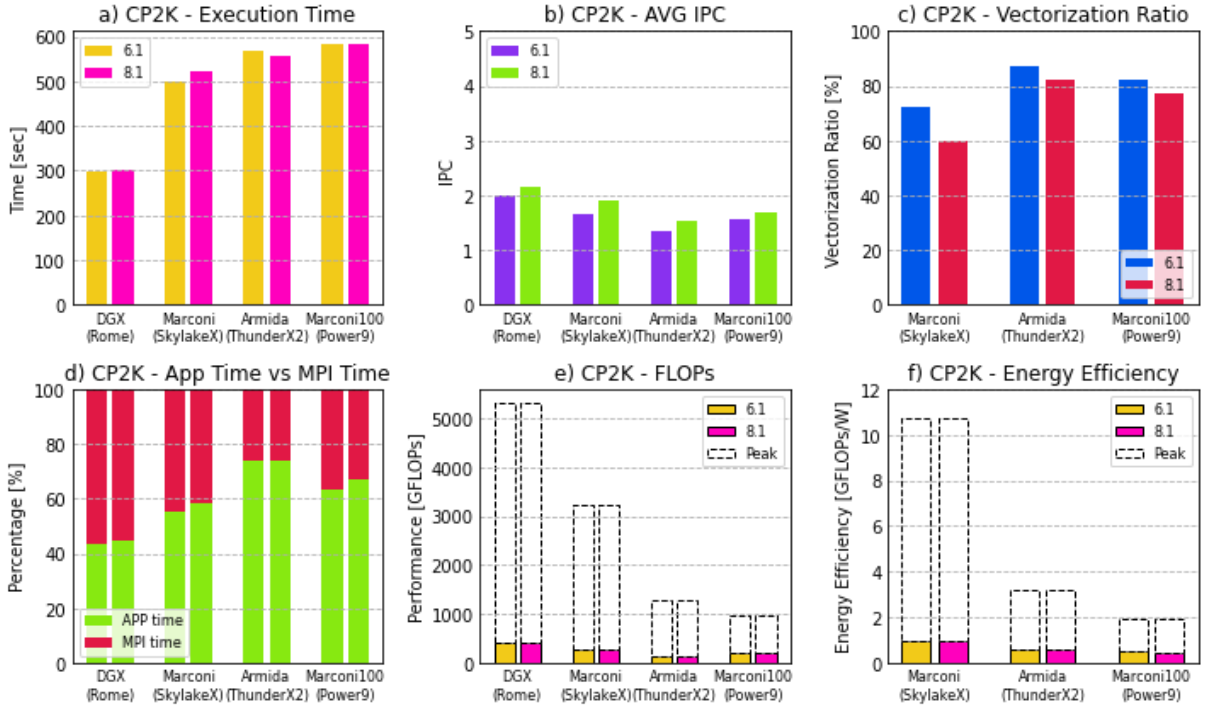

Fig. 13: Experimental results of CP2K 6.1 (pre-MaX project) vs 8.1 (in-MaX project).

In this analysis we don't appreciate any improvement in the performance of CP2K. This is due to the fact that CP2K mostly took benefit from the exploitation of external domain-specific libraries developed in the framework of the MaX project (cfr. WP2). Since in this analysis work we decided to focus on the bare application running on CPU, the improvement coming from the libraries is not noticeable.

2.6.4. BigDFT

BigDFT is another code limited by the memory bandwidth similar to CP2K. But differently from this last one, the trend of the time to solutions in Fig. 14.a has an unexpected improvement for Marconi100. We think that this is given by the capacity of the Power9 to move huge amounts of data within the chip (up to 7 TB/s).

The low IPC, shown in Figure 14.b, for every platform also means a back-end saturation of the microarchitecture of the cores that usually is caused by memory bottlenecks. We can exclude in part

a limitation given by the network communications, for the Fig. 14.d reports a limited activity of MPI time in the application, quantified on average in 22.55% for all runs. If we also consider the low vectorization of the code in all platforms shown in Fig. 14.c, the results seem very poor in terms of microarchitectural performance for FLOPs and energy efficiency reported in Fig. 14.e/f. We have to keep in mind that BigDFT is a code that is highly memory bound as the compute per byte of the workload is very low (see Chapter 2.7. in the roofline mode). Moreover, this code is not able to leverage on scientific libraries with a highly optimized feature for the specific microarchitecture, as vendors do not usually provide accelerated routines for convolution operations.

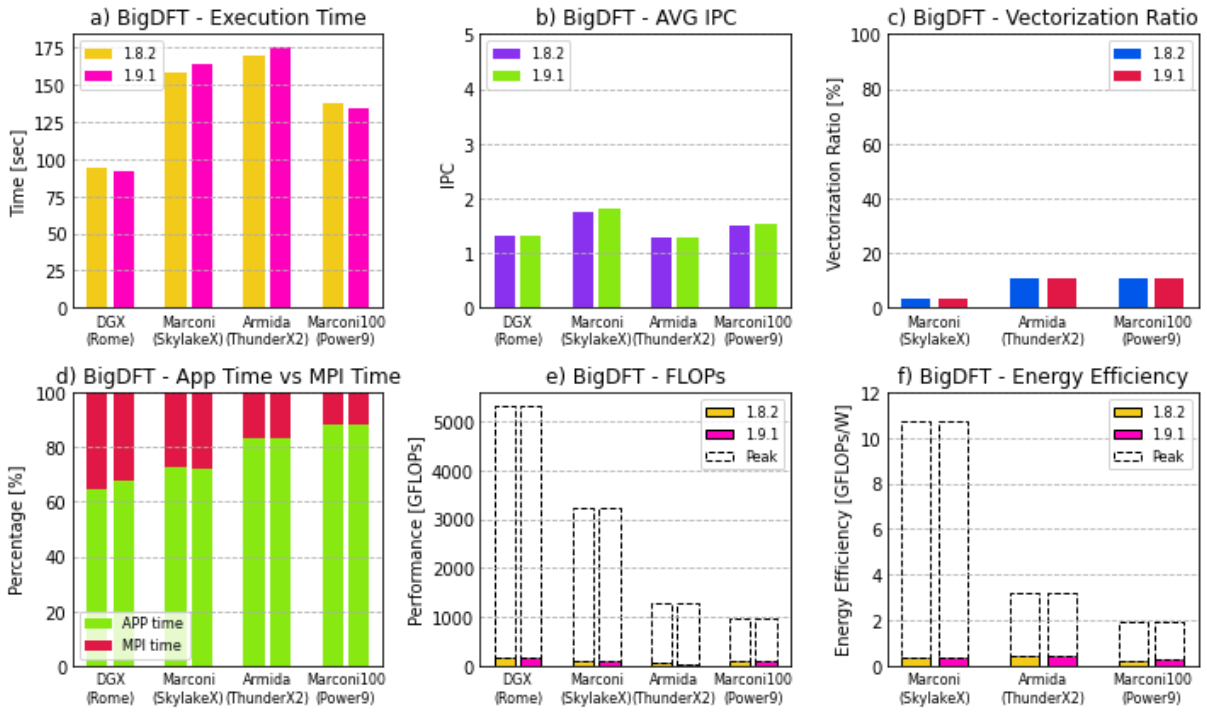

Fig. 14: Experimental results of BigDFT 1.8.2 vs 1.9.1.

In terms of improvements between the 1.8.2 version (pre-MaX project) and the new 1.9.1 version (in-MaX project), we can see on average a slight improvement in time to solution and better utilization of the instruction level parallel of the system thanks to an improvement of the IPC for all architectures.

In conclusion, due to the natural compute per byte of the BigDFT workload, we can say that calculating the energy efficiency using the FLOPs and the average of power consumption of the CPU is not a metric that can represent the efficiency of the code in terms of overall performance and energy efficiency. BigDFT is a code suitable for computing systems with a high data movement capacity (at all levels, from the main memory to the L1 cache) and equipped with low vector instructions to have an effective utilization of the microarchitecture.

2.6.5. Yambo

Similarly to the other MaX codes, Yambo also shows a memory boundness. It is interesting to see that all our platforms have a similar time-to-solution, as shown in Fig. 15.a, even if the architectures have a huge performance gap in terms of FLOPs performance and memory bandwidth.

From Fig. 15.d, we can exclude that Yambo is limited by the MPI communication, the MPI time being the lowest of all MaX codes. In turn, this would also facilitate the scaling of this application. The Yambo code shows behaviours similar to BigDFT due to a low IPC and vectorization ratio shown in Fig. 15.b/c. This is also reflected in a low FLOPs performance and consequently in a low energy efficient performance.

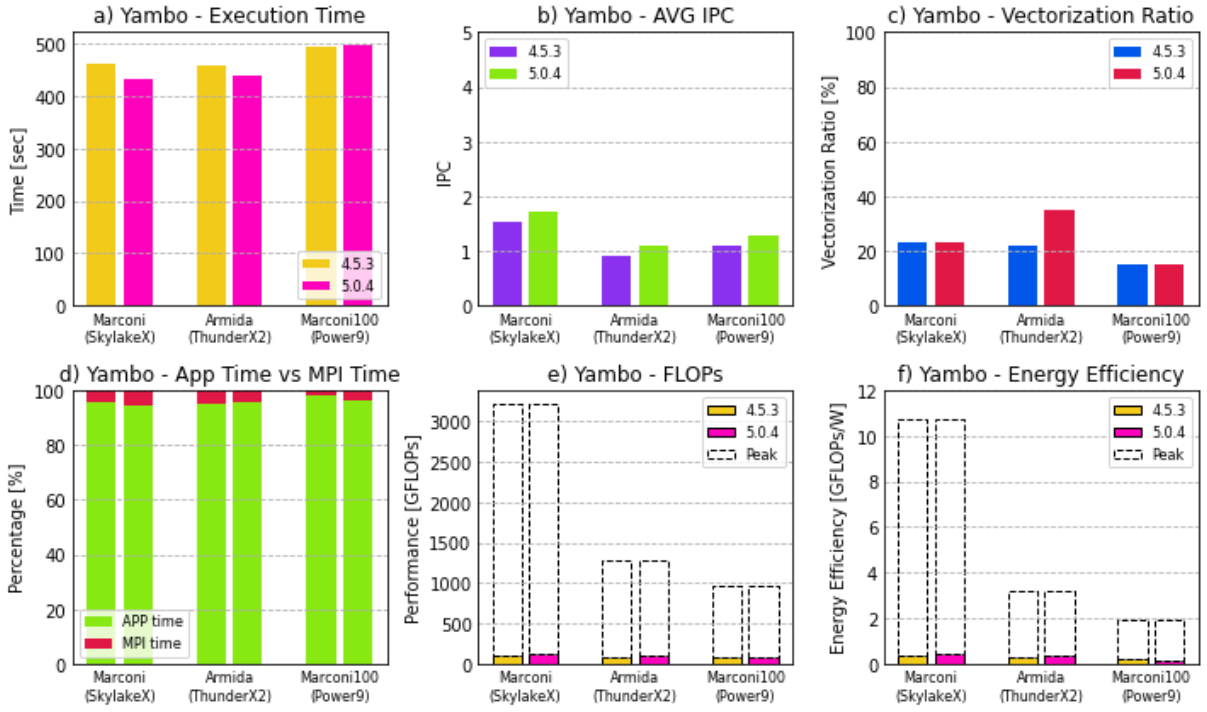

Fig. 15: Experimental results of Yambo 4.5.3 (pre-MaX project) vs 5.0.4 (in-MaX project).

For this specific code, we are not able to show the experimental results of the DGX platform due to some incompatibility with the acceleration libraries. We worked with the code owners in this project to fix these issues but we have not been able yet to add on this analysis for time reasons.

We could conclude that Yambo is limited by the data movement on all our target platforms that are not able to keep their execution units busy. But these experimental results need to be extended with more performance analysis because they are insufficient to show the bottlenecks of this complex code. We leave further performance exploitation in future evolution of the MaX project for it requires a specific analysis only on this code.

2.6.6. Siesta

To conclude our performance analysis, we present the experimental results on Siesta, also shown in Fig. 16.

The first thing we see in the plot 16.a, is the huge performance improvement in the time to solution for DGX and Marconi systems between the 4.0.2 (pre-MaX project) and the MaX-1.3.1 version (in-MaX project). This improvement is given by the new supported ELSI[34] library that is able to accelerate the computation of Siesta. If we look at Figure 16.c, we can see that the vectorization is very high for all the platforms. We only have a drop of vectorization around 50% for the MaX-1.3.1 version in Armida and Marconi100. This is explained by the fact that the ELSI library does not provide accelerated routines for these specific microarchitectures, and this also impacts on the time to solution for both platforms. As we have already seen, compute bound applications limit the IPC around 2 constrained by the number of vector execution units as shown in Fig. 16.b.

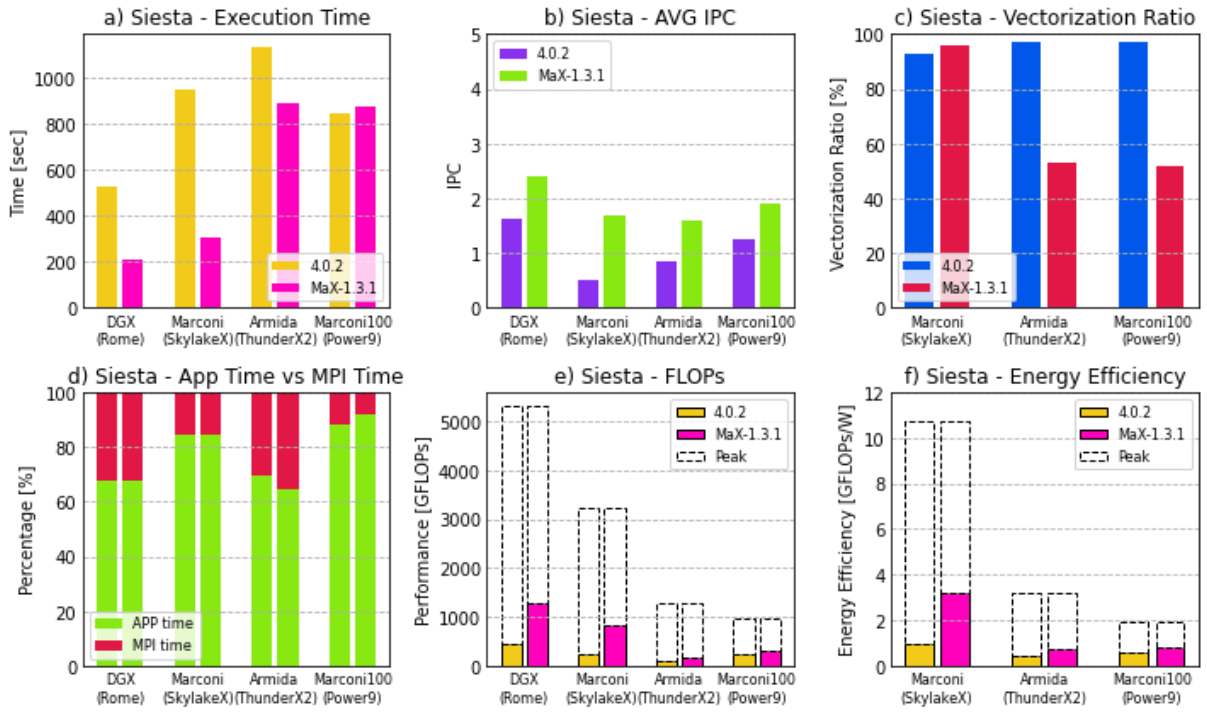

Fig. 16: Experimental results of Siesta 4.0.2 vs MaX-1.3.1.

While application and MPI time ratio are on average with respect to the other codes, the FLOPs and consequently the energy efficiency of Siesta have a very good improvement with the support of ELSI reaching a high overall performance.

This analysis classifies Siesta as a compute bound code but only when optimized libraries are used to accelerate the overall computation. It is interesting to see as a code can become CPU bound only supporting new routines that can change the entire behaviour of the application. One of the

[34] https://wordpress.elsi-interchange.org

outcomes of this analysis is that it is very important to identify optimized libraries and tune properly for the target architecture in order to reach a high level of efficiency.

2.7. TMAM

Having presented the analysis of all the MaX codes, we now want to show some interesting results based on the top-down analysis methodology reported in Chapter 2.1.2 and applied to the Marconi and Armida systems. We had to restrict this analysis to these architectures as AMD Rome and IBM Power9 do not expose all performance counters needed to realize a TMAM for architecture and software stack limitations.

In Fig. 17, we see the experimental results for all the MaX codes. The TMAM analysis shows a very good classification for all codes, which respects the TMAM target for HPC applications. The first interesting result is the difference of the TMAM analysis between Marconi and Armida: we can clearly see that the architecture of Marconi is more limited due to the back-end, and this happens because the MaX codes require a huge utilization of the memory bandwidth since in this architecture the execution units are very powerful. On the other hand, Armida shows a high utilization of the execution units because most of the retaining cycles are spent in executing instructions. This means that the backend, equipped with a large memory bandwidth, is able to supply the execution units spending most of the cycle in the instruction retiring.

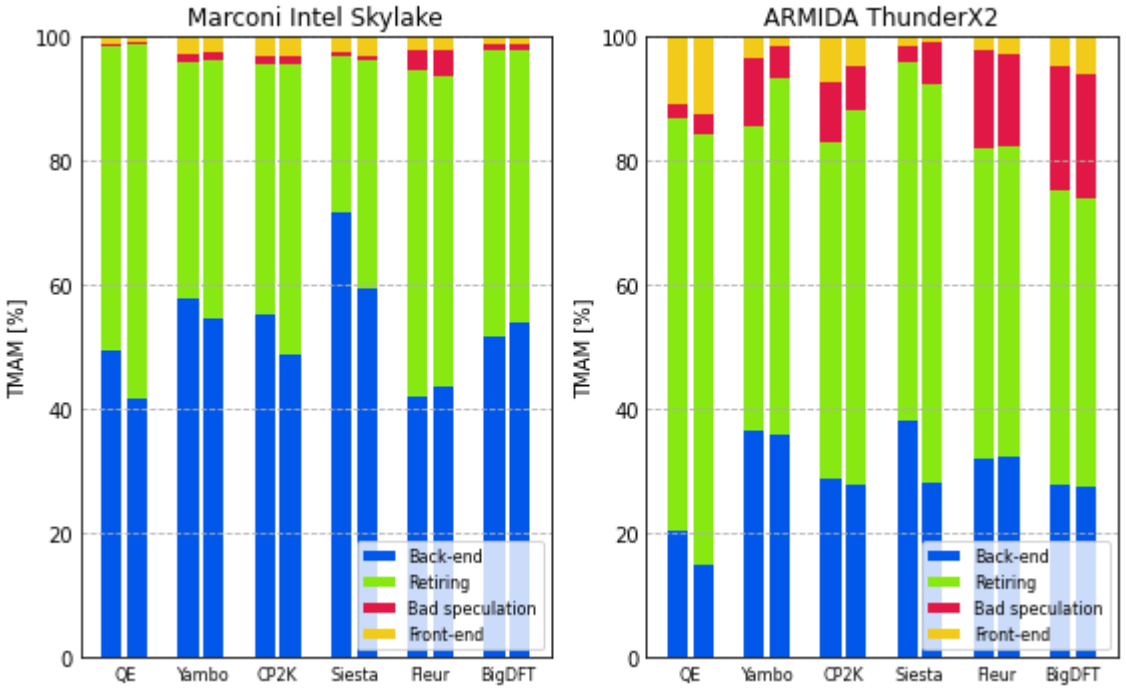

Fig. 17: TMAM analysis of MaX codes on Marconi and Armida systems.

It is also interesting to see that the ThunderX2 architecture lacks performance in speculative executions of the instructions with respect to Intel Skylake architecture. Since the MaX codes are really productive and very complex, they consequently show a complex execution path of the instructions, which induces a bad utilization of the CPU's resources when the speculation is wrong.

For instance, BigDFT on Vulcan architecture shows up to 20.11% of the total instructions executed are not retired, wasting those cycles. This is typically caused by a poor performance of the branch prediction unit (BPU) of the Vulcan architecture also shown in Fig. 5.

In general the MaX codes show a good mix of utilization which is a typical behaviour of HPC applications. The conclusion of this analysis also proves that most of the code needs high performance memory architectures to maintain the execution units busy. Moreover, this analysis also shows the limitation of simple front-end architectures that are not able to intercept complex branch workflows; this can impose important performance penalties. This also explains the lower performance of the MaX codes on this architecture with respect to the other platforms.

2.8. Roofline Model

We now consider a roofline model analysis, shown in Fig. 18, to confirm the memory and compute bound nature of the MaX codes. Unfortunately, we are able to use only the Marconi100 system for this analysis for, as previously explained, we can collect information on the memory bandwidth only on this platform in production systems.

Before commenting on the experimental results, we highlight that the IBM Power9 is a CPU with a high memory bandwidth and a poor performance in terms of peak FLOPs. From this architecture we expect the main limitation to come from the execution units, since the small vectorization would be the bottleneck of the application.

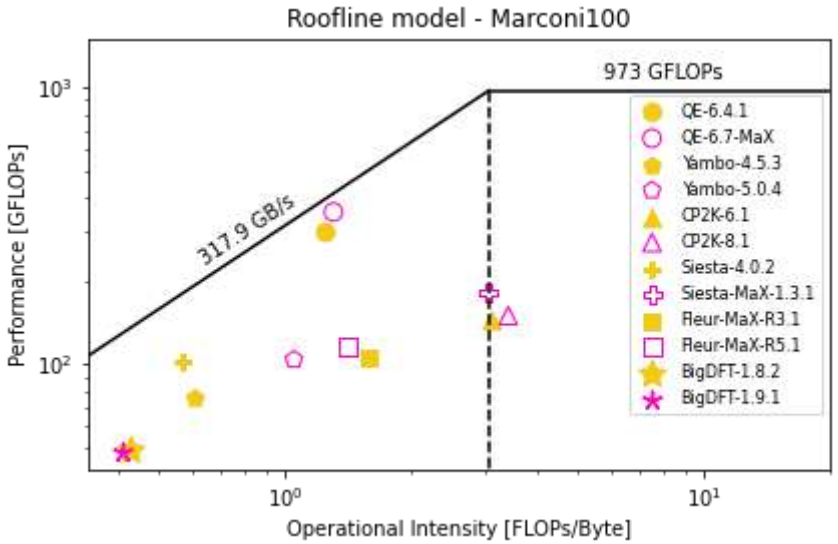

Fig. 18: Roofline model of MaX codes on Marconi100.

However, in Fig. 18 we can clearly see that this is not the case, since most of the codes are still limited by the memory bandwidth. This is an interesting result that supports the fact that the MaX codes are strongly limited by the memory bandwidth even on high memory bandwidth systems with a low performance in terms of FLOPs. The only exceptions are Siesta Max-1.3.1 and CP2K, which are on the edge between memory and compute bound. For these specific codes, the Power9 reveals itself as a good balance architecture, even though it is not able to exploit all the FLOPs available for CP2K is

limited by the MPI communication, and Siesta is not supported by the ELSI library with optimized routines for Power9 architecture. Moreover, as explained in Chapter 2.6.4., we can prove here the poor operation intensity of BigDFT, which limits the utilization of the execution units of the platform.

2.9. Performance and power efficiency at scale with MPI and OpenMP

While our microarchitecture efficiency analysis is mostly focused on a single node, we want to present a small analysis on the energy efficiency of a rappresentative MaX code when we scale out the application on multiple nodes. We select QE for it shows high efficiency in terms of performance and energy on a single node.

In this work we compare the time to solution, the speed up, the average power, and the energy consumption using a weak scaling analysis exploring different MPI and OpenMP (OMP) configurations, to show the impact of both programming models. We also analyze the ratio between the MPI and application time, the MPI communication bandwidth, and the max memory usage from the application.

Differently from the other energy analysis, in this particular case we use both the energy consumption of CPU and DRAM as we don't need to compare energy measurement from different platforms without incurring unfair comparisons.

In Fig. 19, we show the time to solution and the speed up of QE scaling between 1 and 16 nodes on Marconi by varying the configuration of MPI and OpenMP. In detail, we use OpenMP to scale between 1 and 8 threads. When we use 1 OpenMP thread, we run one MPI task per core (48 cores), instead when we use more than 1 OpenMP thread, we run a number of MPI tasks equal to the number of cores divided by the number of OpenMP threads. For instance, if we use 8 OpenMP threads in a 48 cores node we run 6 MPI tasks on each node and so on.

From Fig. 19.c, we clearly see that the scalability is quite good, using 8 nodes is almost linear with the first loss of scalability using 16 nodes (768 cores). It is also interesting to see from Fig. 18.b that the fastest configuration is to use 2 OpenMP threads for 1 node run, and from Fig. 19.c we see that the best speed up using 16 nodes is still with 2 OpenMP threads.

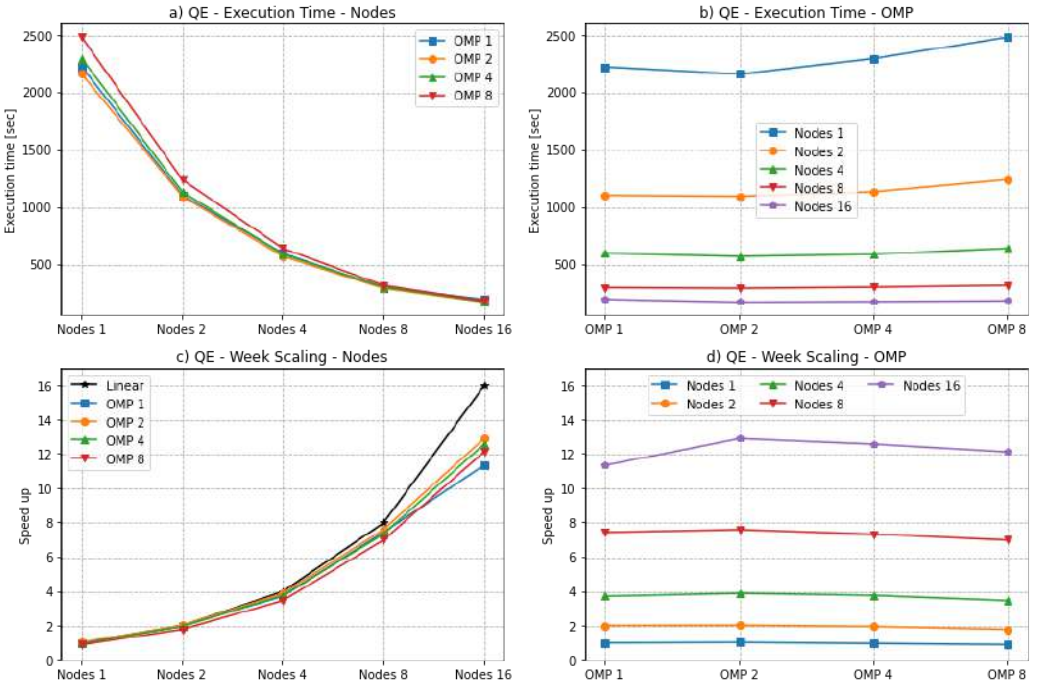

Fig. 19: Time to solution and speed-up of QE in a weak scaling analysis using different OpenMP and MPI configurations on Marconi.

While we are not going to discuss here the scalability of QE in absolute value, we want to focus on the energy consumption and the average power consumption of the system.

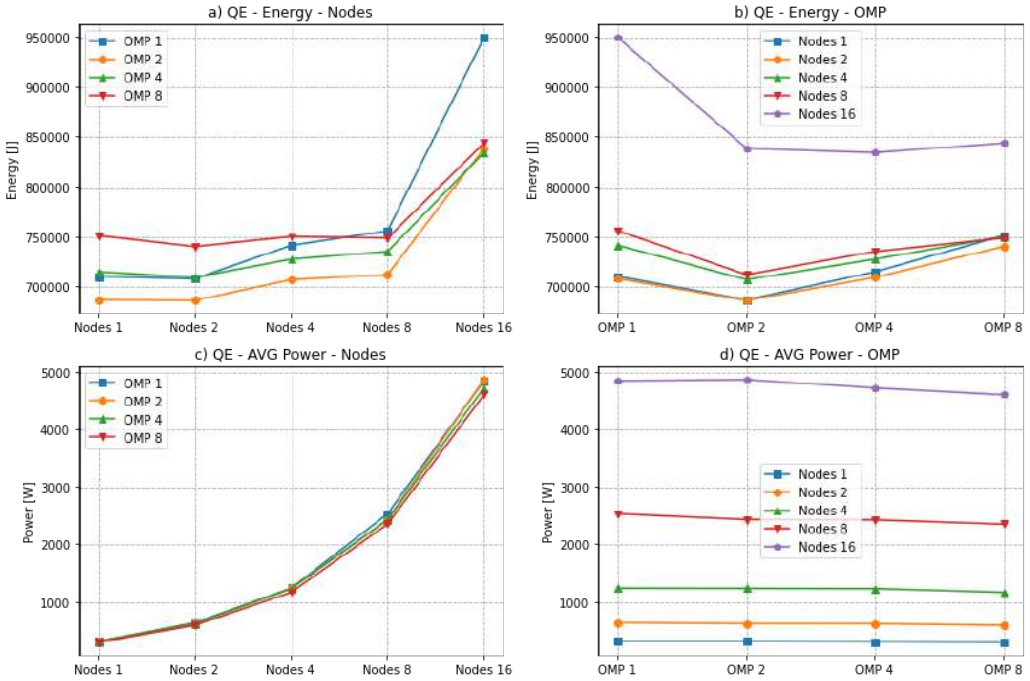

Fig. 20: Energy and average power consumption of QE in a weak scaling analysis using different OpenMP and MPI configurations on Marconi.

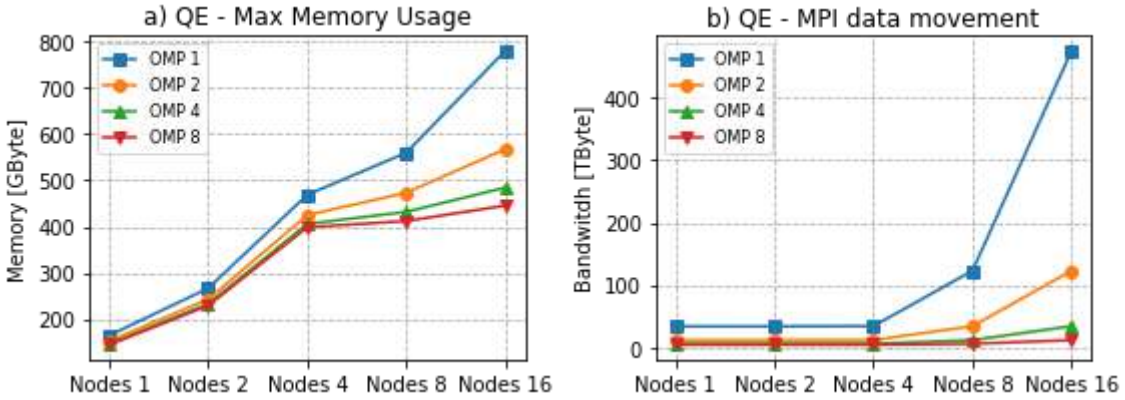

Fig. 21: Max memory usage and MPI data movement of QE in a weak scaling analysis on Marconi.

Fig. 20 shows how the energy and the average power consumption on the weak scaling analysis vary. The first interesting result is to compare the energy with the power in the plot 20.a and 20.c. While we have an energy consumption drop when we use more than 1 thread the average power is still the same, this is probably an effect caused by two factor: i) the first factor is the lower utilization of the HW resources of the system due the OpenMP run only when some parallelism is exposed from the application avoid useless MPI tasks, and ii) the lower number of MPI tasks also induce a lower utilization of the memory as shown in Fig. 21.a. It is also interesting to see from Fig. 21.b how change the amount of data exchanged in QE when the OpenMP is used due the lowering of the MPI tasks and consequently a reduction of the data exchanged using MPI library.

2.10. Packaging manager and code portability

During the project work in the codesign activities, we had to install all MaX codes on several platforms, a task that took a lot of time due to the complexity of the codes that required several scientific libraries to run. The installation of the codes is not a trivial task for users and it is often overlooked by the code developers. For this reason we chose Spack[35], a well-known package manager in the HPC community, to redistribute the MaX codes in a simple way and to make their installation more friendly. Spack is a well-known package manager in the HPC field used by several supercomputing centres. It is an open-source project started from the LLNL supercomputing centre and fastly adopted in the HPC world. It is not only a US Department of Energy project, but it has also been selected from Japan supercomputing centres as the official tool to realize the software stack for their systems. It is taking place very quickly in Europe and several supercomputing centres have started to collaborate within the project to make use of it as a productive tool. At MaX we started to collaborate with the Spack community to create a set of Spack recipes aimed at facilitating the installation of MaX codes. One of its greatest advantages is the possibility to build and install multiple versions and configurations of softwares. It works on Linux, macOS, and supports the software stack of several supercomputers. Moreover, Spack offers a simple interface to specify which dependency libraries to compile within the application (Intel MKL vs OpenBLAS vs Netlib-LAPACK vs etc.) and which version it is possible to use. It also facilitates code portability among different platforms (Intel, AMD, ARM, PowerPC, etc.) using different compilers. For instance, almost all codes used for the

[35] https://spack.io/

performance analysis in this deliverable have been installed with Spack after we contributed to the Spack package available in the project repository[36].

In detail, we have contributed to the Spack project by creating a set of missing installation packages for the MaX project and/or for the dependency libraries used by the codes. We report in the following the current situation of Spack for each MaX code:

- **Quantum ESPRESSO**: for this code we did not submit any package modification as it works out of the box with the current Spack package present in the project. However, we identified a set of missing options to customize the installation of Quantum ESPRESSO needed to improve the default package. The most important is that the current package misses the GPU version installation of Quantum ESPRESSO. We plan to work on this package in the near future.
- **CP2K**: since the current package of CP2K is quite complete, it doesn't need improvement at the moment.
- **Fleur**: this package was missing at all in the Spack project. We submitted an official package developed in MaX to be reintegrated in the project. The pull request done on the official repository of Spack was approved[37].
- **BigDFT**: this package was missing at all in the Spack project, too. We provided the project with a new complete set of packages for BigDFT. The work to develop this package was quite challenging for this code is actually a suite of applications and libraries. Therefore, we had to develop a single package for each component of BigDFT and to submit the 10 packages with the respective libraries that compose this application. This also facilitates the installation of BigDFT, or a part of it, without compiling the entire suite. At the present time, the pull request is under review[38].
- **Yambo**: this code is currently supported in Spack with a very limited package capable of installing an old version of the application. At MaX, we worked on a more complete package with a lot of variants to install Yambo. At the time of writing we are also concluding this activity and we plan to reintegrate soon in the official repository[39].
- **Siesta**: before developing a package for newer Siesta versions, we had to work on the packages of the Siesta dependencies. For this reason, we reintegrated a set of new packages and we updated old packages already present in Spack that did not satisfy the requirements of Siesta. All these packages have been reintegrated and now we are able to develop a package for Siesta, currently a work in progress. See[40] the accepted pull request with the Siesta dependencies.

[36] https://github.com/spack/spack
[37] https://github.com/spack/spack/pull/26631#event-5446201240
[38] https://github.com/spack/spack/pull/26853#pullrequestreview-788734432
[39] https://github.com/nicspalla/my-repo
[40] https://github.com/spack/spack/pull/26489#event-5443574410

3. ARM architecture tests

In the foreseeable future, particularly in Europe, the place and role of the Arm CPUs will become more and more important. Therefore, it is essential to us to recall here at least two main announcements and initiatives :

- Sipearl Rhea processor is going to be based on Arm Neoverse V1 cores (Zeus)
- Nvidia Grace CPU is also going to be based on a future generation of Arm Neoverse cores. It will be used in the next generation Alps supercomputer at CSCS.

Because these two CPUs are very likely to be used by developers and users of the MaX flagship codes in the coming years, and because the two are based on Arm Neoverse cores, it is important to us to evaluate the MaX flagship codes on the current generation of Arm Neoverse cores: Arm Neoverse N1. This is why in this section we propose to first give an update on compilation as a follow up to the previous deliverable D4.4[41], but also to focus and to present recent performance and benchmark results on Arm Neoverse N1 as a complementary work to the one that we did on ThunderX2 and that we presented in the “2. Performance Portability and Energy Efficiency of MaX codes” section.

[41] http://www.max-centre.eu/sites/default/files/D4.4_First%20report%20on%20co-design%20actions.pdf

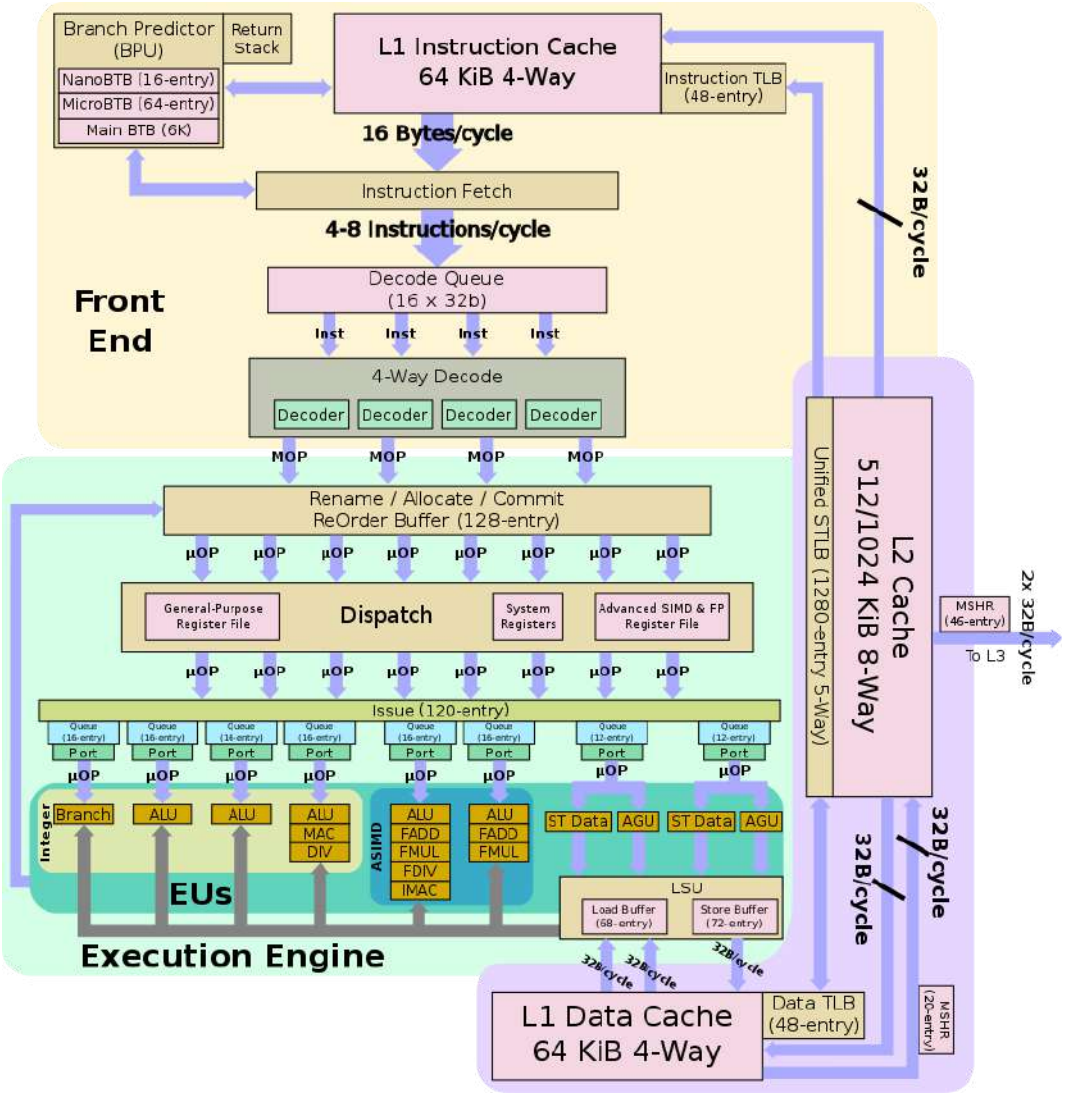

Fig. 22: Overview of the Arm Neoverse N1 microarchitecture[42].

Today, Arm Neoverse N1 cores are available in two CPUs: Ampere Altra and AWS Graviton2. We worked on the AWS Graviton 2 processors. The AWS EC2 c6gn.16xlarge instances were used to take advantage of the full resources of the nodes. The Arm Neoverse N1 microarchitecture on which it is based is presented in Fig. 22. It shares some similarities with the Marvell Thunder X2. The SIMD instructions that both support are NEON, with its 128-bit wide vectors (4 single precision FP elements or 2 double precision FP elements). The cores also benefit from two floating point (FP) units, both supporting FMA instructions. With a clock-speed of 2.5GHz it can perform up to 20 GFLOPs per core. AWS Graviton 2 is a 64 cores single-socket processor without NUMA effects, which means that at the node (or socket here) level it has a theoretical peak performance of 1280 GFLOPs. Furthermore, the

[42] https://en.wikichip.org/wiki/arm_holdings/microarchitectures/neoverse_n1

AWS Graviton 2 features 8 channels of DDR4-3200, that is a theoretical memory bandwidth of 204.8 GB/s.

We started by focusing this work mainly on BigDFT. We believed that the SVE work presented in D4.4 combined with this benchmarking work on Arm Neoverse N1 cores should give us a good understanding of the behavior of BigDFT on future CPUs, such as the Rhea processor from SiPearl. The different use cases and numerical methods led us to perform a similar study for Yambo. Analogous surveys are planned or already started for other MaX Flagship codes.

3.1. BigDFT

Compilation

In the previous deliverable D4.4, we mentioned that BigDFT could be compiled without any issue with GCC 9.2.0 on aarch64. We have used this compiler as well as the 10.2.0 version on various occasions since. No issues were to be reported.
We continued to identify, reproduce, investigate, and report issues faced with the Arm Compiler for Linux (ACfL). The issues were raised upstream so that fixes would benefit the entire community. These issues can be found here :

- https://github.com/flang-compiler/flang/issues/889
- https://github.com/flang-compiler/flang/issues/890
- https://github.com/flang-compiler/flang/issues/1013
- https://github.com/flang-compiler/flang/issues/1014

The issues 889, 890, and 1014 are now closed and fixed. The latest versions of the Arm Compiler for Linux benefit from these improvements. Only issue 1013 is still open.

Benchmark

At the time of the D4.4 report some BigDFT runs and tests were made on aarch64, but performance was not the main focus. We propose here to go through some benchmark work that has been done recently.

Test cases

The kind of computations performed by an application can depend on the test case. Indeed, bottlenecks and hotspots can be very different from one input file to another. For this reason for this benchmark activity we have decided to use three different datasets. To better understand the behavior of Arm based hardware we chose datasets that had already been used on x86.
B80 is a small test case related to Boron with 80 atoms and 120 orbitals. As it can only be used with up to 60 processes, it is particularly well suited for single node tests running quickly[43].

Porphy is a larger dataset related to Porphyrin with 2048 orbitals. It can be used for up to 1024 processes. The input files and reference results can be found at the link[44].

[43] https://gitlab.com/max-centre/benchmarks/-/tree/master/BigDFT/B80
[44] https://gitlab.com/max-centre/benchmarks/-/tree/master/BigDFT/Porphy

Monomer is a bigger and real dataset related to COVID-19. It can be considered as a production simulation. It contains 4879 atoms for 6826 orbitals. The requirements in terms of memory are not to be neglected: above 1128 GB of memory are required for this simulation. The input files and reference results are available at the link[45].

Hardware and Software stack

For this benchmark we have used AWS EC2 instances: the c6gn.16xlarge instances were used for the AWS Graviton 2 processor, a great example of the usage of Arm Neoverse N1 cores. Their x86 counterpart, the AWS EC2 c5n.18xlarge instances, was used for comparison purposes since they are based on Intel Skylake CPUs.

On the software side, we have used GCC 9.2.0 (and also tried with GCC 10.2.0) and OpenMPI 4.1.1. On the aarch64 nodes we have used the Arm Performance Libraries for BLAS and LAPACK, since it is freely available on the c6gn instances. On x86, OpenBLAS was used.

Impact of OpenMP parallelization

Both MPI and OpenMP parallelization are available in BigDFT to take the most of what a CPU has to offer. Therefore we wanted to have a look at the impact on performances.

We used both B80 and Porphy test cases in order to mitigate the impact of the input set. For each dataset we used a certain number of nodes: from 1 and up to 4 for B80 and between 4 and 32 for Porphy. Let us keep in mind that the parallelization is limited by the number of orbitals. Hence, for instance, in the case of B80, which has 120 orbitals, it is better to use only up to roughly 60 processes. Therefore, using two nodes of 64 cores, with 1 OpenMP thread for each MPI task, leads to 128 MPIs which is far too big. Therefore, this is not a configuration we can use. For this reason it does not appear on the upper left plots. So for a given number of nodes we considered different ways to use all the cores available by varying the number of OpenMP threads per MPI ranks. And then for each configuration, for this number of nodes, we measured the elapsed time and we compared only the different elapsed time of different configurations for a given number of nodes by computing the speed-up compared to the configuration with the highest number of threads. For instance, on the plots below, for the c6gn.16xlarge instance, if we consider 2 nodes, we consider the following configurations :

- 64 MPI ranks times 2 OpenMP threads
- 32 MPI ranks times 4 OpenMP threads
- 16 MPI ranks times 8 OpenMP threads.

Then we take the 16 MPI ranks and 8 OpenMP threads configuration as the reference and compute the speed-up of the other settings compared to this one. We do this for each number of nodes. Therefore it does not make sense to compare a speed-up of two configurations with a different number of nodes. The trends between two numbers of nodes can instead be compared.

[45] https://gitlab.com/max-centre/benchmarks/-/tree/master/BigDFT/monomer

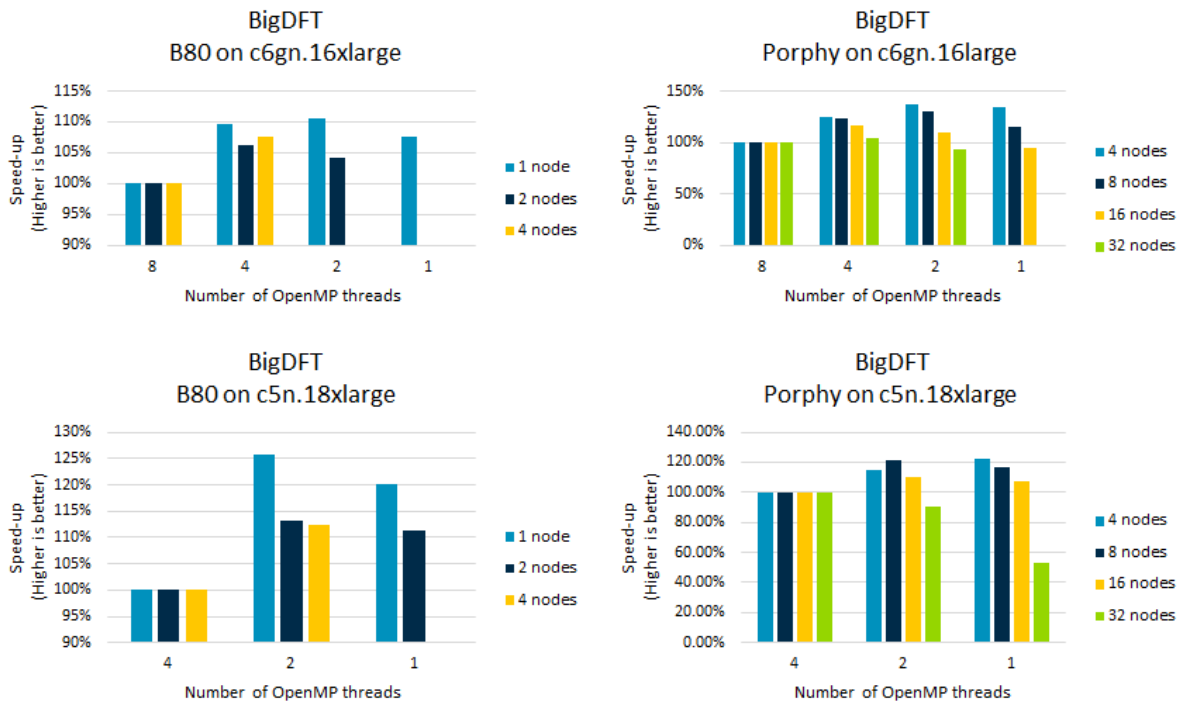

Fig. 23: Impact of the number of OpenMP threads on the performances of BigDFT.

The B80 results show that on the Intel based instances (c5n.18xlarge), using 2 OpenMP threads is the best configuration independently of the number of nodes. Indeed, for a given number of nodes, it brings between 10 and 25%, which is not negligible. With the AWS Graviton 2 it is a bit different. Indeed, for 1 node, using 2 OpenMP threads is the best choice in terms of performance, even though using 4 OpenMP threads offers close performances. But when using 2 nodes it is the other way around: using 4 OpenMP threads is slightly better than using 2 OpenMP threads. In the case of 4 nodes we are limited by the number of orbitals and it is not possible to use 2 OpenMP threads. Using 4 OpenMP threads still brings a fair speed-up (7%) compared to using 8 OpenMP threads though. Therefore, depending on the number of nodes that we are going to use, for BigDFT, we need to adapt the number of OpenMP threads being used. For B80, if we know that we are going to use between 1 and 4 nodes, it appears here that using 4 OpenMP threads would be a wise choice though, because on average, it is going to be the best one.

For Porphy it is more complicated and more interesting. Indeed, on c5n.18xlarge, 1 OpenMP thread leads to the best performance on 4 nodes. But this is not the case at all for the other configurations: 2 OpenMP threads for 8 nodes, 2 OpenMP threads for 16 nodes and 4 OpenMP threads for 32 nodes. Choosing to use 1 OpenMP thread at scale because it leads to the best performance on 4 nodes would be a mistake with a significant loss of performance at scale (32 nodes). We see that for BigDFT a given number of OpenMP threads would not work at any scale to get good performance. We see similar behaviors on c6gn.16xlarge for Porphy. We need to adapt it to the number of nodes, cores, and to the dataset. In other words, it needs to be adapted to the workload, to the number of orbitals per MPI rank. The trend on Porphy would suggest that at scale using more OpenMP threads is going to be beneficial. Probably because it will decrease the number of MPI ranks and therefore increase

the number of orbitals per MPI rank. Each MPI rank needs to have a certain amount of work. Using a high number of OpenMP threads is probably not needed when not using a lot of nodes because the number of MPI ranks is already low (directly connected to the number of nodes) and therefore the number of orbitals per MPI rank is high enough.

In conclusion, we can say that the usage of OpenMP in BigDFT is important. It can lead to significant performance improvements. From our experiments it appears that the best number of OpenMP threads that needs to be used depends on the workload on each node. Therefore, we need to adapt it to the test case (number of atoms and orbitals) as well as the resources available (nodes and cores). We could say that the best configuration depends on the number of orbitals per MPI rank.

B80 results

In this part we want to present and discuss the results of the BigDFT benchmark for the small dataset B80. We started the performance study by having a look at the scalability properties of BigDFT. The main purposes are to understand the behavior of BigDFT at scale but also compare it on two different platforms x86 and aarch64. As discussed in the previous section, we have adapted the number of OpenMP threads for each number of nodes and cores in order to get the best performances. Hence, for a given number of nodes and cores we present only the best performances (lowest elapsed time).

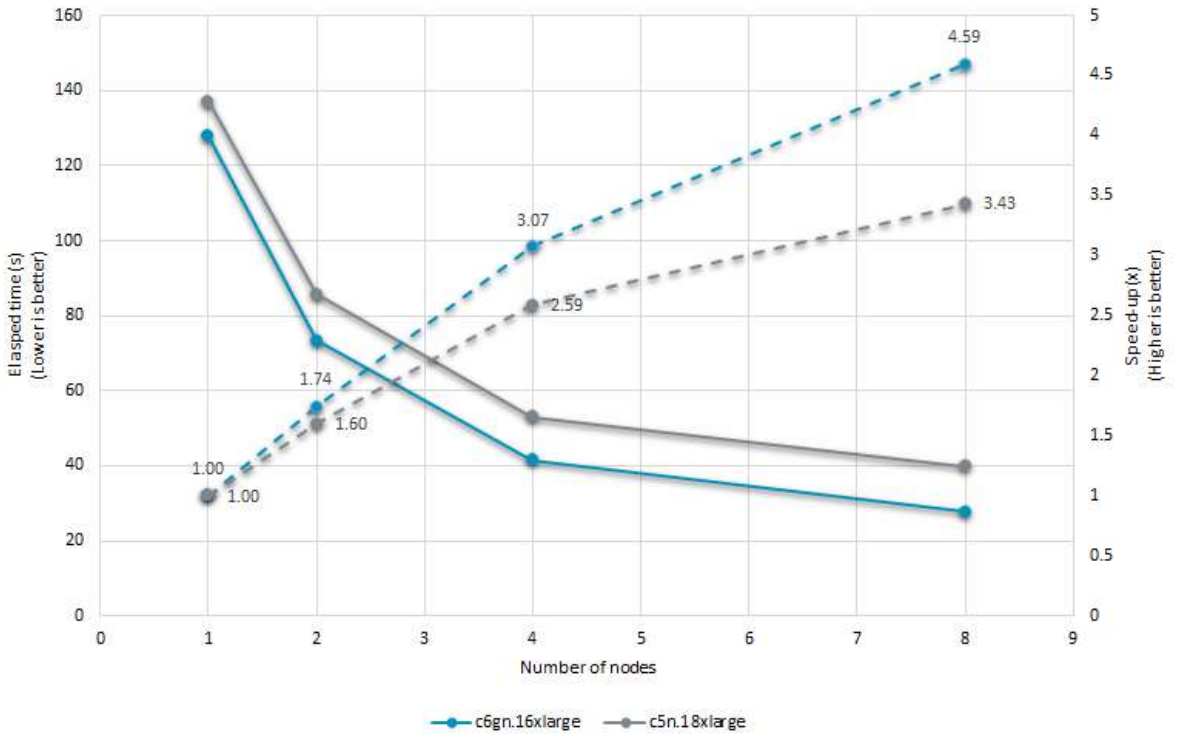

Fig. 24: Strong scalability study of BigDFT for the B80 dataset.

For this small testcase, we took one node as the reference for the elapsed time and used up to 8 nodes. We can see that when increasing the number of nodes the elapsed time decreases, which

means that BigDFT leverages the additional resources. Nonetheless, ideally the elapsed time should be divided by two each time the resources are multiplied by two. The parallel efficiency on two nodes is good: at least 80%, but it decreases quickly: between 42% and 57% at height nodes. The size of the test case could explain it if most of the time is spent in the communications. The results for the bigger test cases should enable us to have a better understanding of this. The behavior on the two platforms is similar: no major differences can be seen in terms of scalability between x86 and aarch64.

Now that we have had a look at the scalability of BigDFT for B80, we want to evaluate the performance on Arm Neoverse N1 cores. We start by comparing the performances at the core level and especially the elapsed time for a given number of cores. As we can see, the elapsed time on Arm Neoverse N1 cores is very close to the one on Intel Skylake cores. In other words, the per-core performances are similar between Arm Neoverse N1 (c6gn.16xlarge) and Intel Skylake (c5n.18xlarge). This means that thanks to mature software stacks (compilers and libraries), BigDFT can be used on Arm Neoverse N1 cores and will be able to leverage what it has to offer in terms of performance in the same way as it would on x86. Furthermore, thanks to the scalability properties of BigDFT (discussed in the previous section), if what matters for the user is time to solution, using the highest number of cores available will lead to the best time to solution.

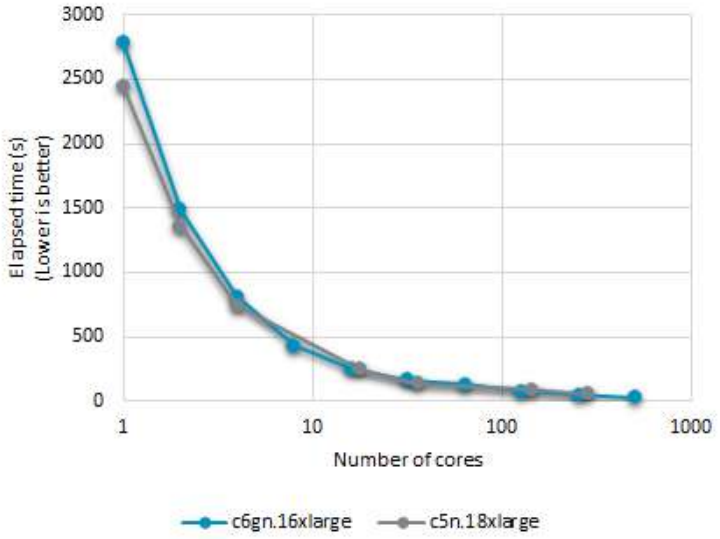

Fig. 25: Comparison of the performances of BigDFT at the core level on c5n.18xlarge and c6gn.16xlarge for the B80 dataset.

In HPC, we tend to use all the resources available in one node, therefore, it makes sense to compare performances at the node level. This time we can see that the gap in terms of performances for a given number of nodes is bigger. Indeed, thanks to its higher core count (64) the c6gn.16xlarge outperforms the c5n.18xlarge (36 cores) on average by 24% . This means that density is important for BigDFT. It also shows that for BigDFT, the current generation of Arm Neoverse-based hardware is competitive against x86 solutions.

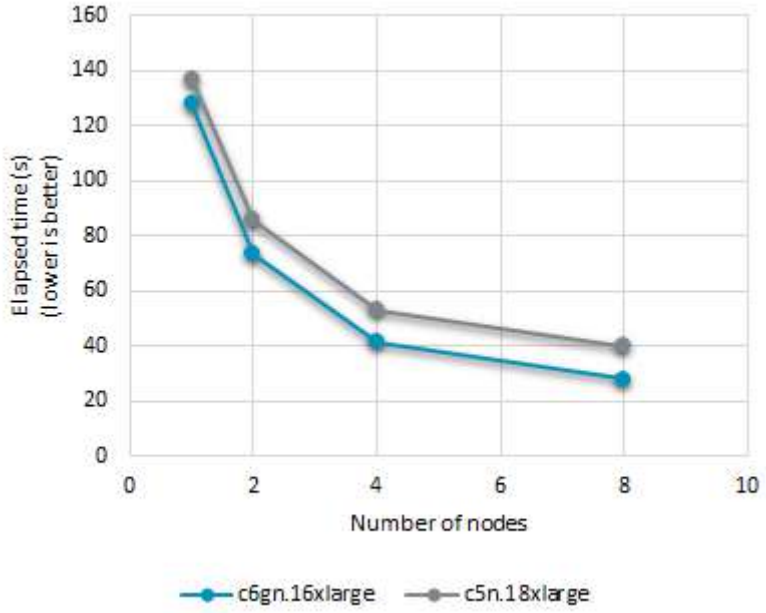

Fig. 26: Comparison of the performances of BigDFT at the node level on c5n.18xlarge and c6gn.16xlarge for the B80 dataset.

Finally, we thought that it would be interesting to have a look at the cost of the simulations on the different platforms. One can see that the Arm Neoverse N1 based c6gn.16xlarge offers a fair benefit compared to the x86 c5n.18xlarge, since, on average, BigDFT simulations on c5n.18xlarge are 74% more expensive than on c6gn.16xlarge. Furthermore, if what matters for the end user is not the time-to-solution, that is to say, if the end user can wait to get the results, but the cost of the simulation matters, the lowest number of nodes will offer the best price-to-solution.

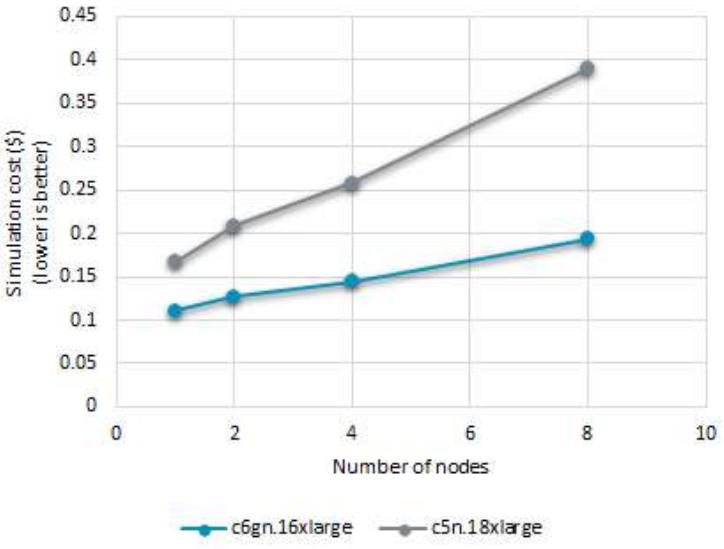

Fig. 27: Comparison of the simulation cost of BigDFT on c5n.18xlarge and c6gn.16xlarge for the B80 dataset.

Porphy results

We propose in this section to discuss the scalability and performance results of BigDFT for the Porphy test case in the same way that we did for B80.

For the small dataset B80, BigDFT was able to take advantage of additional computational resources but the curve was flattening quickly showing a rapid decrease of the parallel efficiency. For Porphy, which is a bigger test case, we can also see that for both c6gn.16xlarge (AWS Graviton 2 - Arm Neoverse N1 cores) and c5n.18xlarge (Intel Skylake) the elapsed time decreases when the number of nodes increases, which once more shows the capability of BigDFT to leverage computational resources. This time, the scalability curve flattens less and the parallel efficiency is higher. BigDFT has good scaling behaviour on both x86 and aarch64 platforms.

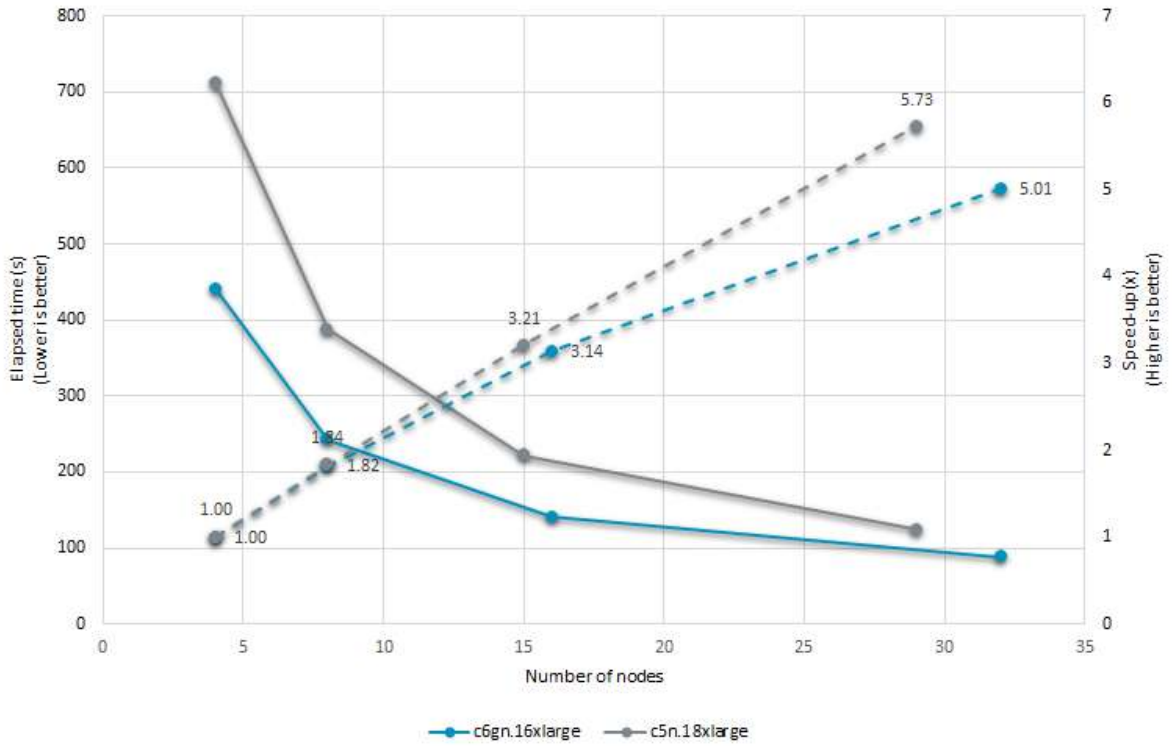

Fig. 28: Strong scalability study of BigDFT for the Porphy dataset.

Now that the scalability properties of BigDFT are confirmed, we have a look at the performances of BigDFT for the Porphy testcase. For B80 the time-to-solution performances on a per-core basis were very close between Arm Neoverse N1 cores and Intel Skylake. We can see that this is also the case here: the blue (c6gn.16xlarge) and grey (c5n.18xlarge) are very close to each other and almost overlap.

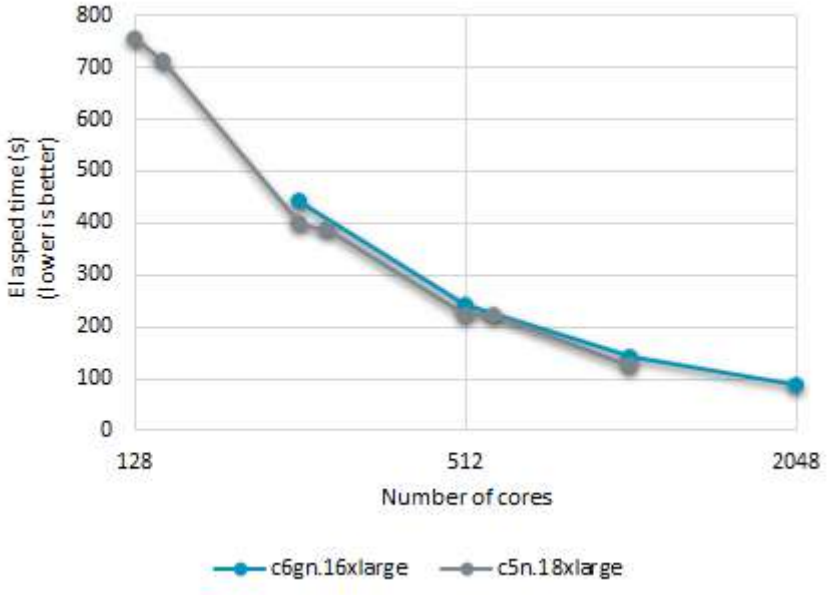

Fig. 29: Comparison of the performances of BigDFT at the core level on c5n.18xlarge and c6gn.16xlarge for the Porphy dataset.

Per-node-performances are still on average better on AWS Graviton 2 than on Intel Skylake. This confirms what we noticed for the B80 test case.

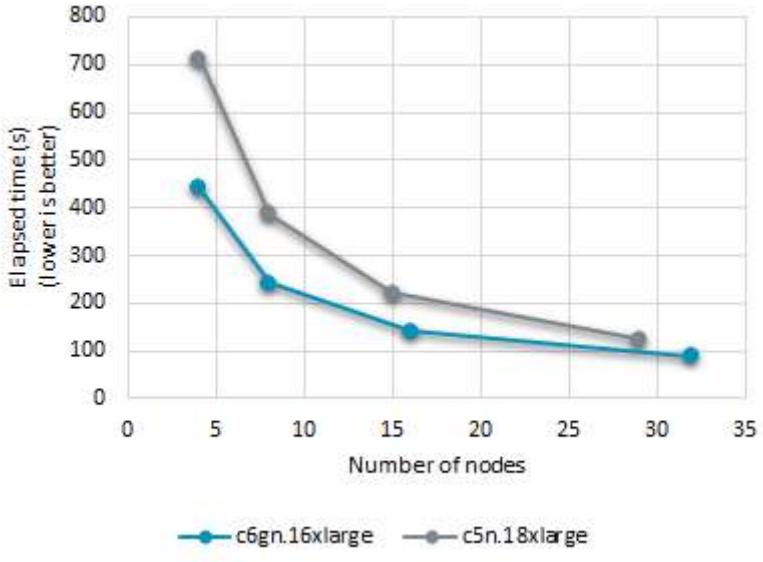

Fig. 30: Comparison of the performances of BigDFT at the node level on c5n.18xlarge and c6gn.16xlarge for the Porphy dataset.

For B80 AWS Graviton 2 was outperforming c5n.18xlarge instances in terms of price-to-solution. This is still the case for Porphy. On average, c5n.18xlarge is 2x more expensive than c6gn.16xlarge.

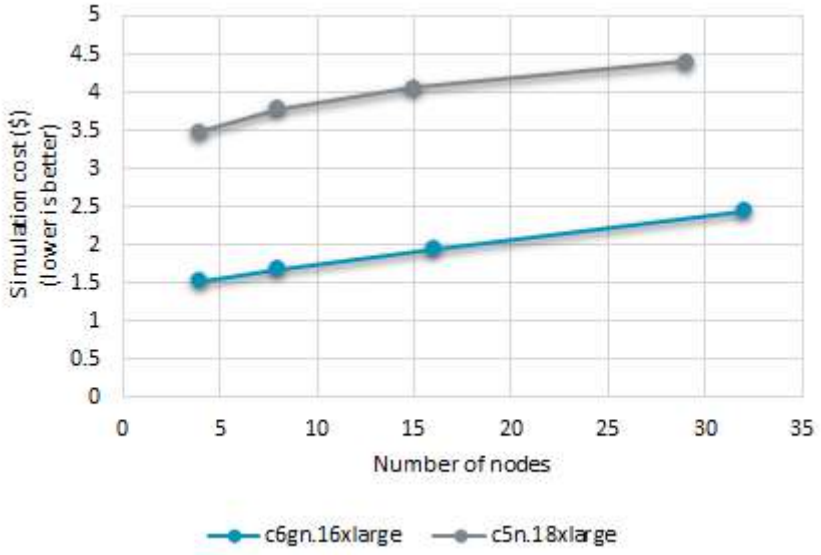

Fig. 31: Comparison of the simulation cost of BigDFT on c5n.18xlarge and c6gn.16xlarge for the Porphy dataset.

Monomer results

We have been able to show in the previous sections that BigDFT performs as well on Arm Neoverse N1 based hardware than on x86 thanks to benchmark studies on small (B80) and large (Porphy) datasets. This time we explore the performances for a real test case: Monomer, that relates to the study of COVID-19.

The production runs were done on two machines : the AMD Rome partition of Irene at TGCC, and the #1 in the TOP500 list (at the time of the simulations) Fujitsu A64FX based Fugaku at Riken. We propose here to run the same simulations on a cluster of AWS EC2 m6g.16xlarge (AWS Graviton2) instances to understand how the Arm Neoverse N1 cores perform.

As we did for B80 and Porphy, we started by having a look at the time-to-solution on a per-core basis. For B80 and Porphy, the performances for a given number of cores were very similar between AWS Graviton 2 (Arm Neoverse N1 cores) and Intel Skylake. We can see that the difference between AWS Graviton2 and AMD Rome is slightly bigger here. On average, for a given number of cores the AMD Romes are 16% faster. If we take a look at the hardware specifications, the frequencies are roughly the same: 2.5GHz for AWS Graviton2 and 2.6GHz for the AMD Rome of Irene. Hence the difference is unlikely to come only from here. On the computational side of things, the vector units of AMD Rome are twice larger than the ones of AWS Graviton 2, therefore only having a 16% gap of performance on a per-core basis shows that the vector units might not be used at their best. On the memory bandwidth side of things, the AMD Rome nodes are bi-socket and therefore benefit from more

memory channels at the node level than the single socket AWS Graviton2. Once more we can be surprised by only having a 16% difference in terms of performances on a per-core basis since AMD Rome nodes benefit from bigger computational power and bigger memory bandwidth capabilities. We can consider that the Arm Neoverse N1 cores are able to be competitive here and BigDFT is probably able to get a higher percentage of the theoretical peak performance of the AWS Graviton2 compared to the AMD Rome.

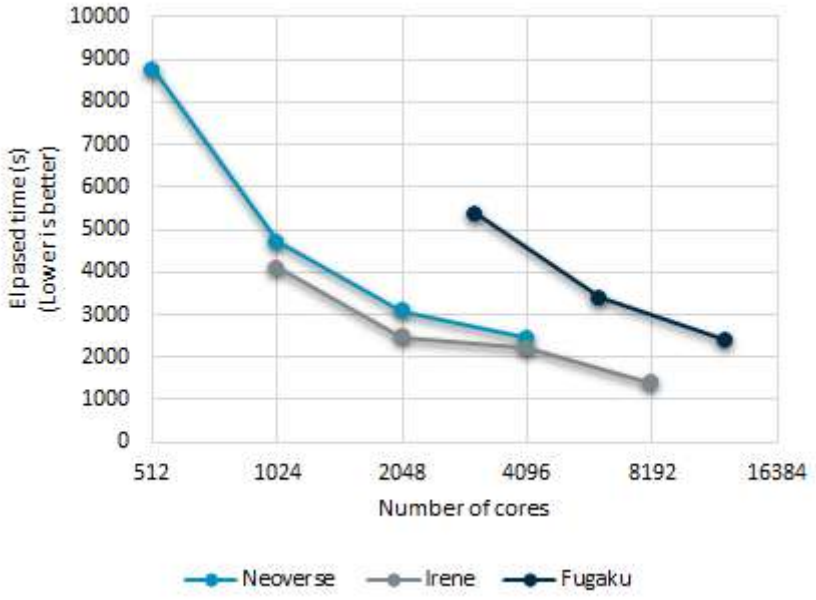

Fig. 32: Comparison of the performances of BigDFT at the core level on three different platforms for the Monomer dataset.

On a per-node basis, on average, the bi-socket AMD Rome offers a 79% performance benefit against the single socket AWS Graviton2. This can be explained by the higher number of cores at the node level for the AMD Rome (64 cores per socket, bi-socket) compared to the AWS Graviton2 (64 cores per socket, mono-socket).

It seems to us that these results prove that codes like BigDFT can be used today on Arm Neoverse N1-based hardware with competitive performances. This is encouraging as regards the performances of future generations of Arm Neoverse cores that will go inside Sipearl Rhea or Nvidia Grace.

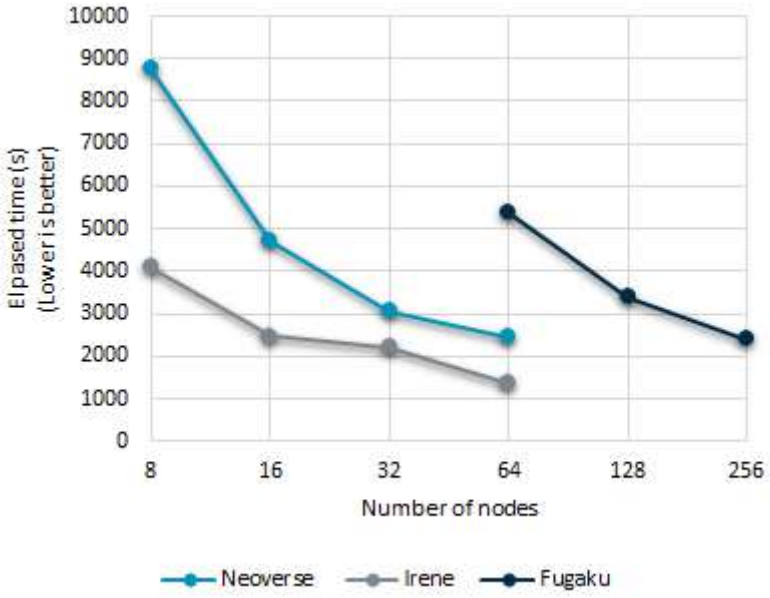

Fig. 33: Comparison of the performances of BigDFT at the node level on three different platforms for the Monomer dataset.

Something that we have not yet discussed so far are the performances on the A64FX nodes of Fugaku. On Fugaku more nodes were used compared to Irene or AWS Graviton 2 instances. This is because of the memory requirements for such a dataset. Unlike the AMD Rome or AWS Graviton2 that use a few hundreds of gigabytes of DDR, the A64FX nodes only have 32GB of HBM2 per node. This is a limiting factor here and therefore we need more nodes in order to be able to perform the simulation. This is something interesting in a co-design approach: the computations performed by BigDFT require a certain amount of RAM per node, because this can be a limiting factor, which is not necessarily the case for other kinds of simulations and workloads.

In terms of performance, the simulations on A64FX are slower than on Arm Neoverse N1 or AMD Rome. We have done a high level profiling study to understand where this gap comes from. In order to counter the fact that the number of cores per node is different for the platforms, we decided to present the "core seconds" (elapsed time multiplied by number of nodes multiplied by number of cores per node) instead of using the raw "elapsed time". We here present the results for 64 nodes.

From the plots, we can see that there is an obvious hotspot on A64FX. Indeed, the time spent in "FLIB lowlevel" is suspicious compared to the one spent on AMD Rome or AWS Graviton2. Since these are some optional utilities, we were able to improve the performances on A64FX. Another aspect is the fact that the A64FX is the first CPU that features the Arm SVE ISA. Therefore some work on libconv has been done to better leverage this SIMD ISA and to improve the performances on A64FX and future CPUs that will use SVE like Sipearl Rhea. This specific work is presented in the first part of the D4.4[46] report and in the D4.5[47] report on code profiling and bottleneck identification. Nonetheless,

[46] http://www.max-centre.eu/sites/default/files/D4.4_First%20report%20on%20co-design%20actions.pdf
[47] http://www.max-centre.eu/sites/default/files/D4.5_Final%20report%20on%20codes%20profiling%20and%20bottlenck%20identification.pdf

some tuning work remains to be done to better leverage the capabilities of the A64FX. Decreasing the time spent in the "unknown" category could also lead to a fair performance speed-up.

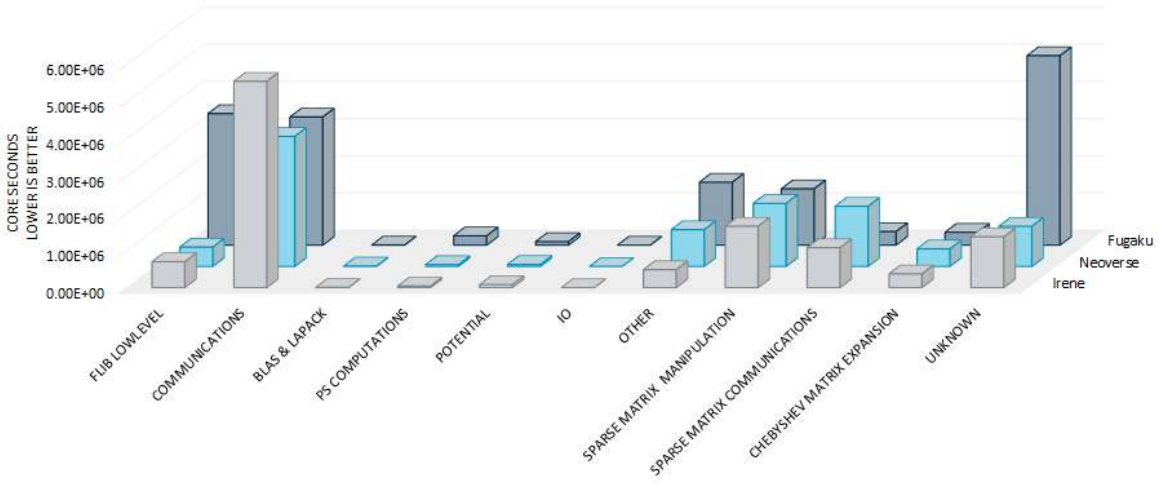

Fig. 34: High level profiling of BigDFT for 64 nodes on three different platforms for the Monomer dataset.

We have shown that the current generation of Arm Neoverse cores N1 is already competitive today to run BigDFT simulations. This is encouraging as regards the performance on future CPUs like Sipearl Rhea or Nvidia Grace that will use future generations of Arm Neoverse cores.

3.2. Yambo

In the previous section we benchmarked BigDFT on AWS Graviton2 and compared it to results on Intel Skylake and AMD Rome to evaluate the Arm Neoverse N1 cores. The results were encouraging and we propose to do something similar for another MaX flagship code, Yambo.

hBN 2D

We started with the small test case mentioned in a tutorial[48]. We evaluated both GCC and Arm Compiler for Linux. For GCC the performances were very similar to the reference ones (see tutorial) :

[48] http://www.yambo-code.org/wiki/index.php?title=GW_parallel_strategies

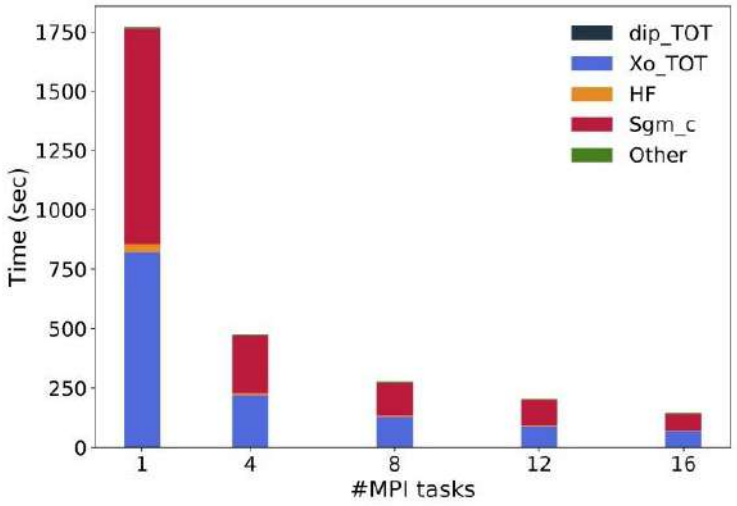

Fig. 35: Reference results of Yambo on x86 for the hBN 2D dataset.

But with the Arm Compiler for Linux the performances were pretty poor.

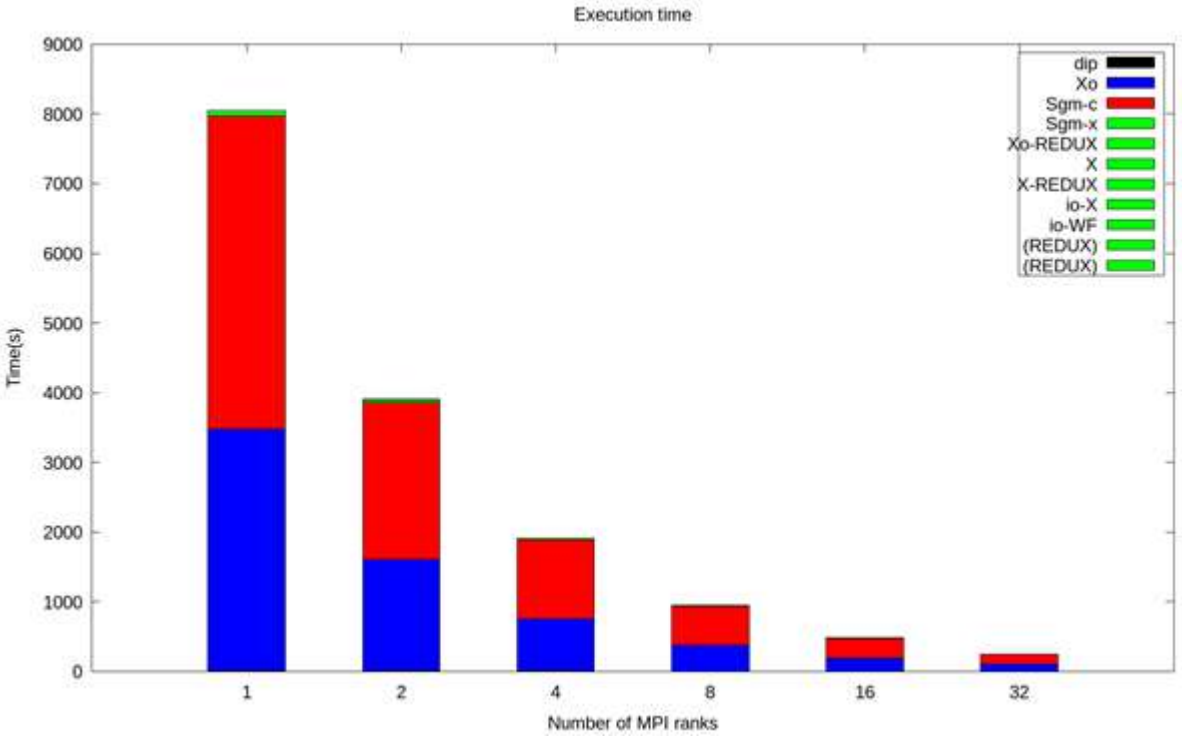

Fig. 36: Timings of Yambo compiled with Arm Compiler for Linux on aarch64 for the hBN 2D dataset.

By tuning the compilation flags we were able to better the performance by 2x, but it was still slower than GCC and the reference results on x86. Hence we did a profiling study to understand the origin of the lack of performance.

depth	Total core time	Function(s) on line	Position
0	100.00%	main [program]	
1	100.00%	main	main.f90:2
2	100.00%	qp_ppa_cohsex	main.f90:111
3	62.40%	__mth_i_cdiv	QP_ppa_cohsex.f90:43
4	19.90%		
4	2.80%	4 others	
5	0.80%		mod_functions.f90:6
5	0.70%		mod_functions.f90:16
5	0.70%		mod_functions.f90:17
5	0.60%		
3	32.90%	__mth_i_cdiv, __mth_i_cdiv@plt	QP_ppa_cohsex.f90:42
4	0.80%		
4	0.10%		
3	3.40%	__mth_i_cdiv	QP_ppa_cohsex.f90:41
4	1.30%		
4	0.80%	1 other	
5	0.80%		
3	1.30%	3 others	
4	0.70%		mod_functions.f90:24
4	0.40%		QP_ppa_cohsex.f90:47
4	0.20%		QP_ppa_cohsex.f90:46

Fig. 37: Profiling of Yambo compiled with the Arm Compiler for Linux on aarch64 for the hBN 2D dataset.

We quickly identified the function performing the complex division as the source of the issue. We made a reproducer and escalated the issue to the compiler team of Arm. A fix was done and integrated in the Arm Compiler for Linux.

rutile-H

Having used a small test case, we then propose to use a larger one also used for benchmarking on x86 and for accelerators, rutile-H[49]. We proposed to act as for BigDFT, that is to say, to compare performances on two kinds of AWS EC2 instances: c6gn.16xlarge (AWS Graviton 2 with Arm Neoverse N1 cores) and c5n.18xlarge (Intel Skylake).

On the software side of things, we used GCC 9.2.0 and OpenMPI 4.1.1 on both platforms and let Yambo compile the libraries it needs (BLAS, LAPACK, ScaLAPACK, LibXC, HDF5 and NetCDF). We compiled Yambo with and without OpenMP. After a few tests it did not appear obvious that we would get better performances on AWS Graviton 2 when using multiple OpenMP threads. Therefore we only used MPI for the parallelization.

For one c6gn.16xlarge node the number of MPI ranks was decomposed in the input file as follows:

[49] https://gitlab.com/max-centre/benchmarks/-/tree/master/Yambo/rutile-H

X_and_IO_CPU= "1 1 1 16 4" # [PARALLEL] CPUs for each role

X_and_IO_ROLEs= "q g k c v" # [PARALLEL] CPUs roles (q,g,k,c,v)

DIP_CPU= "1 16 4" # [PARALLEL] CPUs for each role

DIP_ROLEs= "k c v" # [PARALLEL] CPUs roles (k,c,v)

SE_CPU= "1 4 16" # [PARALLEL] CPUs for each role

SE_ROLEs= "q qp b" # [PARALLEL] CPUs roles (q,qp,b)

When increasing the number of nodes, only c for X_and_IO_ROLEs, c for DIP_CPU and b for SE_ROLEs were increased proportionally.

For c5n.18xlarge we did exactly the same thing, the only difference is that we only have 36 cores per node against 64 on c6gn.16xlarge. Therefore the reference input parameters for one node will be :

X_and_IO_CPU= "1 1 1 9 4" # [PARALLEL] CPUs for each role

X_and_IO_ROLEs= "q g k c v" # [PARALLEL] CPUs roles (q,g,k,c,v)

DIP_CPU= "1 9 4" # [PARALLEL] CPUs for each role

DIP_ROLEs= "k c v" # [PARALLEL] CPUs roles (k,c,v)

SE_CPU= "1 4 9" # [PARALLEL] CPUs for each role

SE_ROLEs= "q qp b" # [PARALLEL] CPUs roles (q,qp,b)

As done for BigDFT, we wanted to have a look at the scalability of Yambo and identify whether the behaviour is different on aarch64 compared to x86. We took as the reference elapsed time the one for two nodes and then increased the resources. Ideally when increasing the compute resources by two the execution time should be divided by two and therefore the speed-up should be 2x.

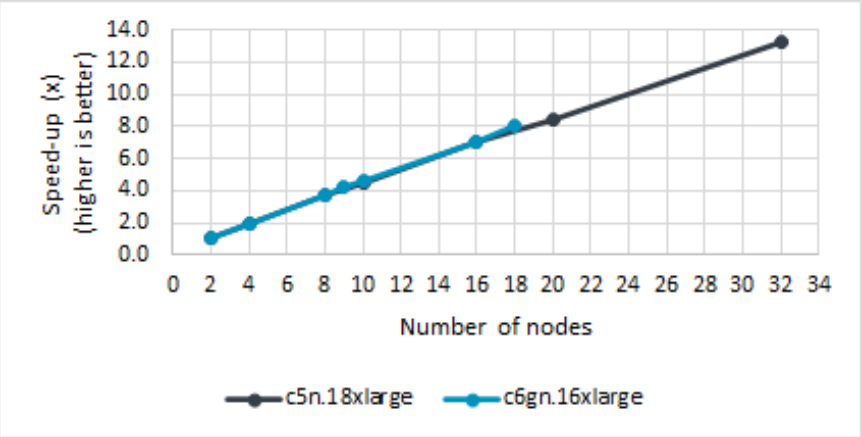

Fig. 38: Strong scalability study of Yambo on two different platforms for the rutile-H dataset.

First, it appears clearly here that the two scaling curves, respectively blue for AWS Graviton 2 and grey for Intel Skylake perfectly overlap. This shows that we do not see different scaling behaviors on the two different platforms.
Now as regards the scalability itself, we can easily notice that the curves are almost straight lines showing an almost ideal speed-up. Indeed at the beginning we can see that when using 4 nodes instead of 2 we have a speed-up of 2 and then when using 8 nodes a speed-up of 4, which is exactly the ideal speed-up. For 18 nodes of c6gn.16xlarge we have a speed-up of 8, which is very close to the ideal one of 9. And for 32 nodes of c5n.18xlarge we have a speed-up of 13.5x, which is still very close to the ideal one of 16x.
In this example the scalability of Yambo is impressive. It would be interesting to check if the same behaviour can be seen for other datasets. It could also be interesting to further increase the computational resources to identify the point for which the scalability starts decreasing. Furthermore, Yambo behaves as well on Arm Neoverse N1 based hardware than on Intel Skylake.
Now that we have had a look at the scaling behavior of Yambo for the rutile-H dataset, let us focus on the performances themselves. On a per-node basis, the performances of the c6gn.16xlarge instances are close to the ones of the c5n.18xlarge. Indeed the blue and grey curves almost overlap. To be more precise, on a per-node basis, c6gn.16xlarge is 8% slower than c5n.18xlarge.

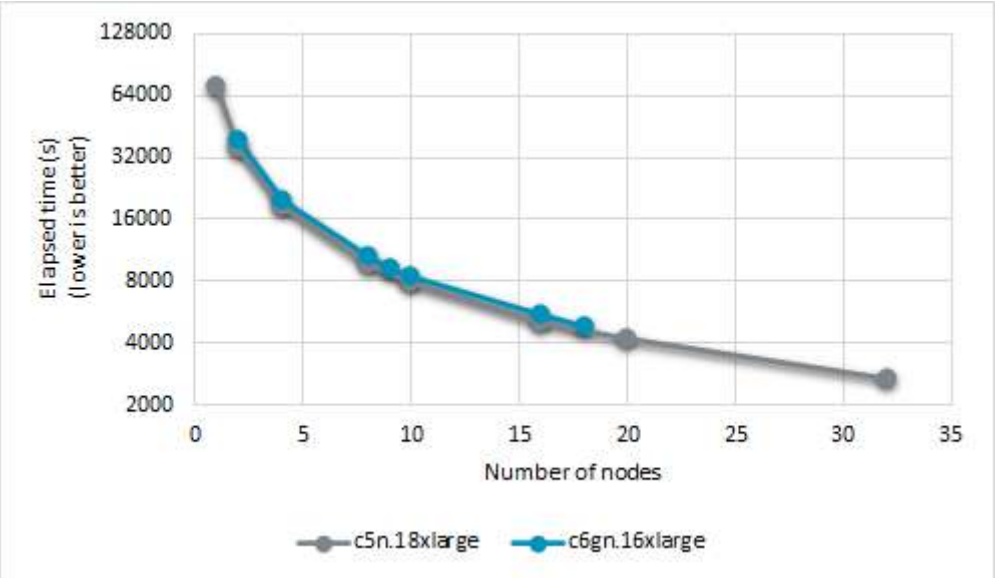

Fig. 39: Performance comparison of Yambo at the node level between c5n.18xlarge and c6gn.16xlarge for the rutile-H dataset.

Now, at a per-core level, the situation is not exactly the same. Indeed the blue (c6gn.16xlarge) and grey (c5n.18xlarge) do not overlap with each other anymore. We are seeing a real difference between the two cores. On a per-core basis, the c6gn.16xlarge is a bit less than 1.8x slower than the c5n.18xlarge. This is interesting because it is very different from what we saw for BigDFT. It would be interesting to perform a fine profiling work, as well as a vectorization study, on both platforms to understand the performance difference. A couple of reasons could explain this gap. First, at a per-node level, because the c5n.18xlarge is bi-socket, the memory bandwidth it offers is bigger than

the one of the single socket c6gn.16xlarge. And as we have less cores per node on c5n.18xlarge, when we compare the performances for a given number of cores, in fact the number of nodes required for the c5n.18xlarge is far bigger than the number of nodes of c6gn.16xlarge instances. This means that for a given number of cores the total memory bandwidth available is bigger on the c5n.18xlarge cluster than on the set of c6gn.16xlarge. This is as regards memory bandwidth. As far as the computational power is concerned, a few parameters could also explain the difference, such as the higher frequency of the c5n.18xlarge and the wider vector units, up to 512 bits with AVX-512, for the c5n.18xlarge compared to 128 bits NEON units on c6gn.18xlarge.

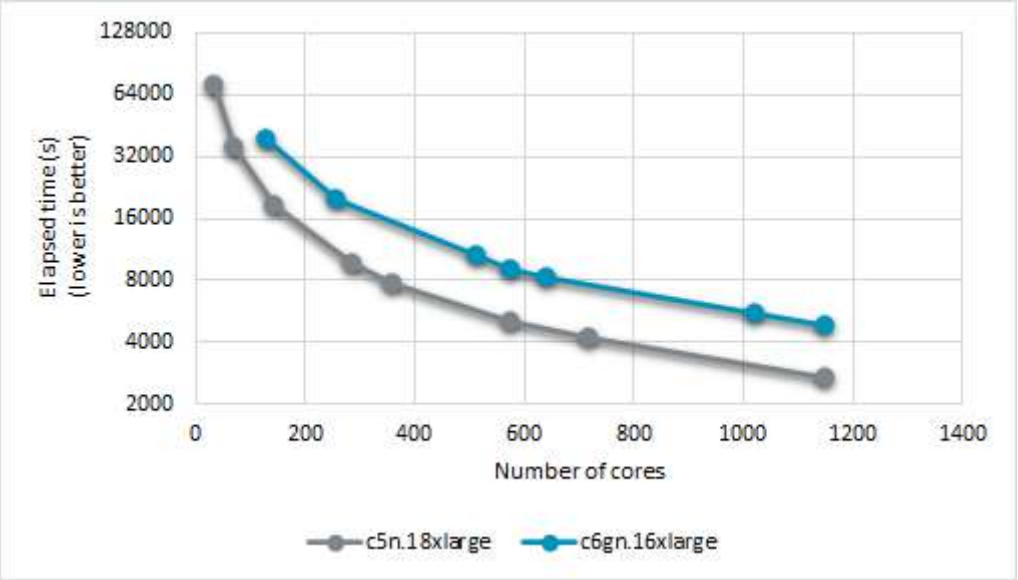

Fig. 40: Performance comparison of Yambo at the core level between c5n.18xlarge and c6gn.16xlarge for the rutile-H dataset.

If we have a look at the high level profiling available with Yambo, it looks like the “Dipole” part is the hotspot for this test case. The inner part of the Dipole calculations is governed by skinny ZGEMM called several times. Matrix-matrix multiplications are known to take advantage of vectorization. If this is the case here and able to leverage AVX-2 or AVX-512 on c5n.18xlarge it would explain the difference in terms of performance between the two kinds of instances. Thinner profiling and vectorization study work would give us answers. If the wider vectors of the Intel Skylake were indeed the source of the better performances, it would be encouraging for the next generation of Arm Neoverse cores, V1. Indeed it will include the SVE instruction set with a vector width two times larger (256 bits) than what is available today on Arm Neoverse N1 (128 bits).
The fact that at the node level we are only seeing a 8% slowdown shows that the higher core count of the c6gn.16xlarge counter-balances the per-core results. This shows that density also matters for Yambo.

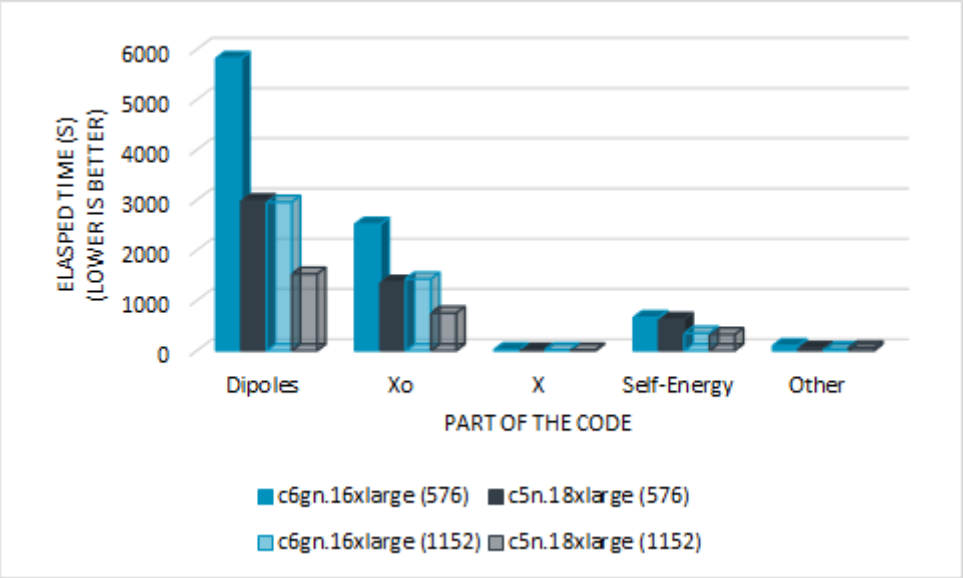

Fig. 41: High level profiling of Yambo on two different platforms for the same number of cores.

Thanks to the high level profiling of Yambo, we can see that all the parts of the code scale well, which was expected in views of the previous scalability plot. We can also see that for a given number of cores, all the parts of the code are roughly 2x slower on c6gn.16xlarge.

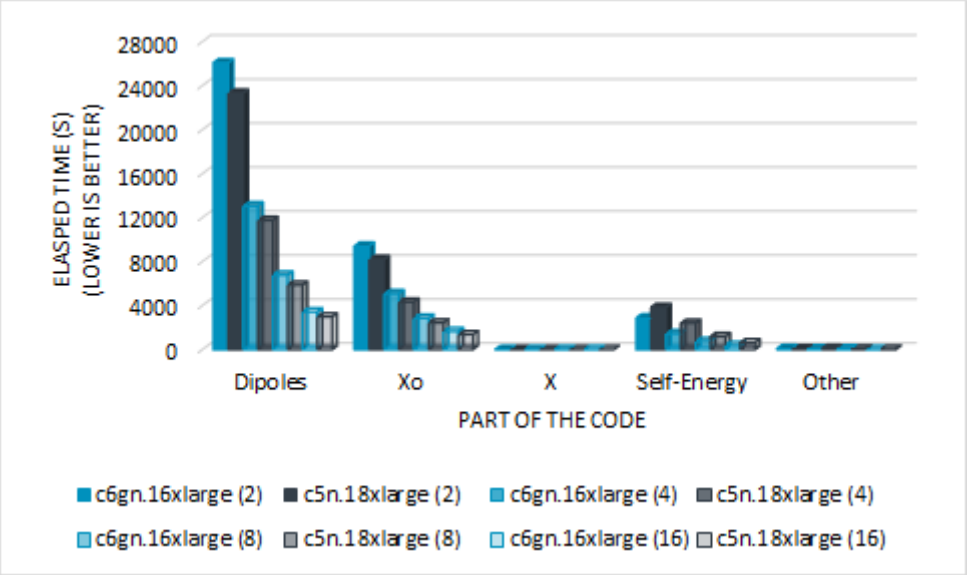

Fig. 42: High level profiling of Yambo on two different platforms for the same number of nodes.

If we get back to the node-level, we can see that the performances are close on the two platforms. The “Dipoles” and “Xo” parts of the code are slightly slower on c6gn.16xlarge. Maybe a tuning of the compiler flags could improve the performance. This is something that could be explored as well as the profiling and vectorization study work that we mentioned. It is also interesting to see that the “Self-Energy” part is slightly faster on c6gn.16xlarge.

The two profiling plots show that the proportions spent in the different parts of the code are similar on the two platforms though. This is encouraging. In our opinion, it would have been suspicious to have a completely different repartition of the time spent in the different parts of the codes between the two CPUs.

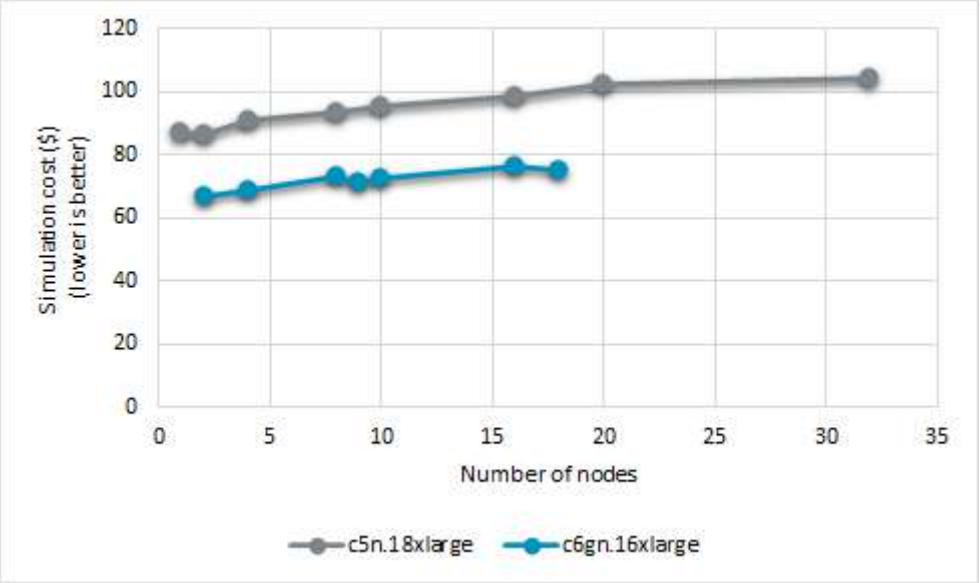

Fig. 43: Cost to solution of Yambo on two different platforms for the rutile-H dataset.

We also quickly had a look at the comparison of the two platforms focusing on the price-to-solution. Similarly to what we observed for BigDFT, the cost of a simulation would be 1.3x more expensive on c5n.18xlarge and would take roughly the same time. Thanks to the good scalability properties of Yambo, the cost of using more nodes to get the solution faster is not far from the cost of using the minimum number of nodes.

3.3. SpFFT

Just like we did for BigDFT and Yambo, we propose in this part to study the performance of SpFFT on Arm Neoverse N1 cores.

We used the development branch[50] of SpFFT to benefit from the already available support for the Arm Performance Libraries (ArmPL). This enabled us to easily build SpFFT with ArmPL. Therefore we first focused on comparing the performances of SpFFT when using ArmPL to the ones when using FFTW. We also tried to explore different MPI and OpenMP configurations to have a look at the scalability.

[50] https://github.com/eth-cscs/SpFFT/tree/develop

We used the following parameters for the benchmark :
benchmark -d 225 343 512 -r 20 -e compact -p cpu -t r2c -s 0.75

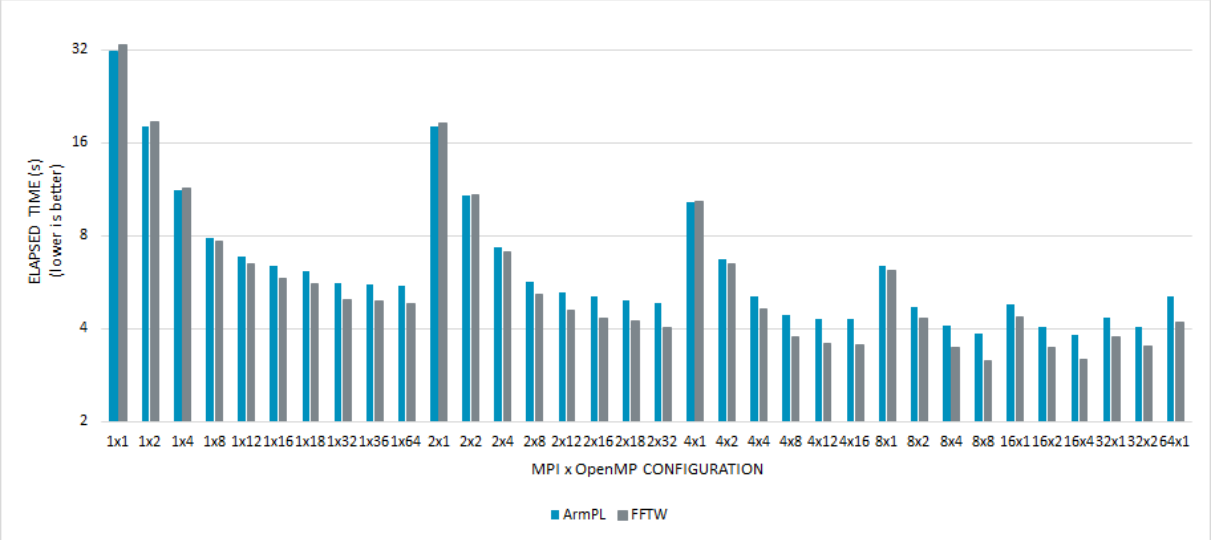

Fig. 44: Performance comparison of SpFFT on aarch64 when compiled with two different libraries: FFTW and ArmPL.

We can first notice that for the input parameters taken for the benchmark the performances when using FFTW are slightly better than when using ArmPL. The difference between the two is small though.
As we did for BigDFT and Yambo, we also compared the performances of SpFFT on AWS EC2 c6gn.16xlarge (AWS Graviton2) and c5n.18xlarge (Intel Skylake). We used the same software stack, GCC 10.2.0 and FFTW.

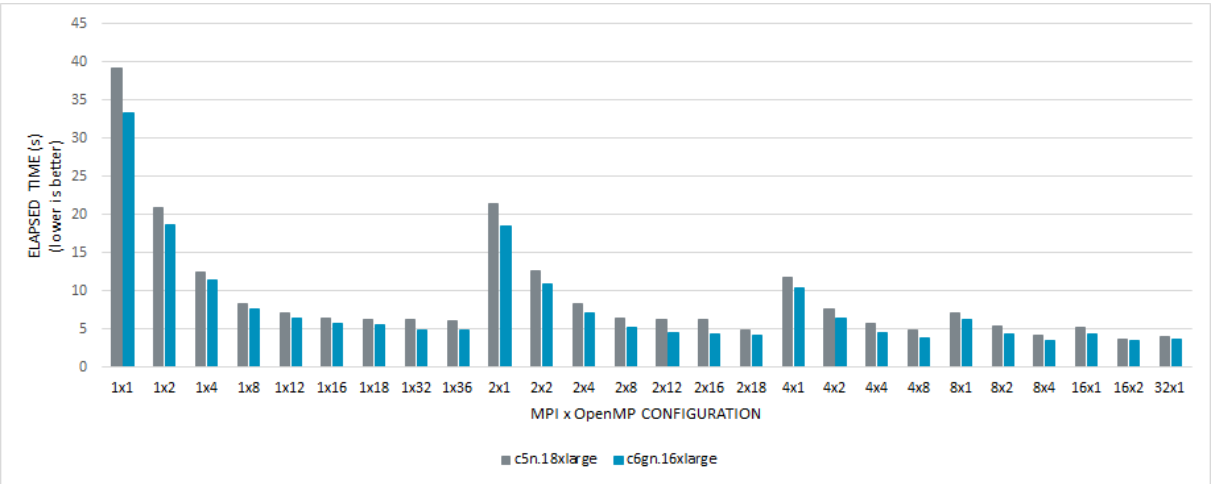

Fig. 45: Performance comparison of SpFFT on c5n.18xlarge and c6gn.16xlarge.

We can see that the difference in terms of performances on the two platforms is not big.

Nonetheless, on average, on c5n.18xlarge SpFFT is 19% slower compared to c6gn.16xlarge. We can see that such a difference is seen for all the MPI and OpenMP configurations, which means that it is not something coming from the higher number of cores at the node level. Therefore we can conclude that for this set of input parameters and for this software stack the performances at the core level are slightly better on AWS Graviton 2.
In the previous plots the impact of the OpenMP and MPI configuration on the performances was also noticeable. Let us focus on c6gn.16xlarge for three numbers of cores: 16, 32, and 64.

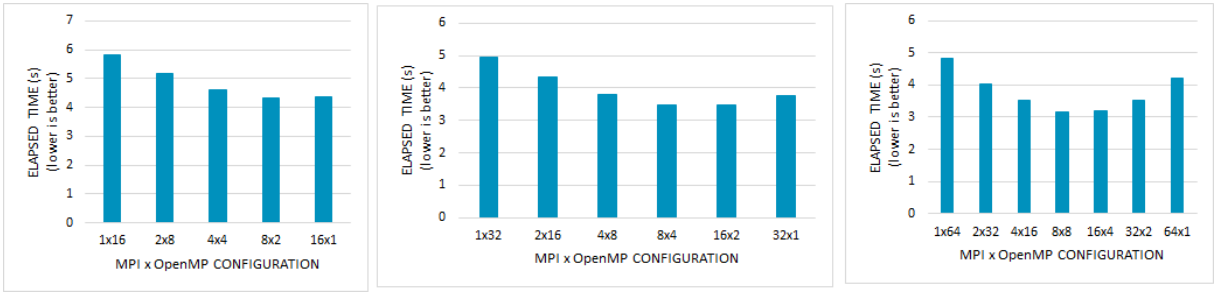

Fig. 46: Impact of the number of OpenMP threads on the performances of SpFFT on c6gn.16xlarge for 16 (left), 32 (center), and 64 (right) cores.

On the left-most figure, for 16 cores, we can notice that using full OpenMP, that is to say 1 MPI rank and 16 OpenMP threads, gives the poorest performance. This is something that we also see for 32 and 64 cores. Still on the left-most figure, it would appear that using 1 OpenMP thread per MPI rank and 16 MPI ranks gives the best performance. This is not the case when using 32 cores (Figure at the center) nor 64 cores (Figure on the right). This means that only using OpenMP or only using MPI does not lead to the best performance in general. The right balance between MPI ranks and OpenMP threads needs to be found in order to get the best performance. This may depend on the workload. Indeed for 16 cores it would appear that using 1 or 2 OpenMP threads is the best configuration. For 32 cores using 2 or 4 OpenMP threads looks better. And for 64 cores using 4 or 8 OpenMP threads is the best configuration. This is a behaviour that we also noticed for BigDFT in the context of the Porphy dataset.

SpFFT looks working as well on c5n.18xlarge and c6gn.16xlarge. This shows the competitiveness of the current generation of Arm Neoverse cores for libraries like SpFFT, but also the capacity of libraries like SpFFT or applications like BigDFT or Yambo to leverage the underlying hardware. This is encouraging for the future generations of Arm Neoverse cores and the CPUs that will be based on it like Sipearl Rhea or Nvidia Grace.

4. MFAT and memory profiling

In this section we will describe a novel tool for memory profiling that has been designed, prototypically implemented, and evaluated within this project.

4.1 Introduction and motivation

Memory architectures of processor architectures are expected to become increasingly complex. Today the majority of processor architectures used for HPC systems are still based on DDR technology, which tends to be a capacity-optimised technology with data rates improving at a rather low rate compared to the increasing compute capability. Faster memory technologies like graphics memory or HBM are now available, but due to costs and technical constraints, architectures that use such technologies feature relatively low capacity. One option to meet both memory performance and capacity requirements is to integrate different memory technologies. A recent example of this approach is Knights Landing (KNL)[51], a many-core processor architecture that supported both DDR4 as well as MCDRAM memory technologies for realising a large-capacity as well as a high-performance memory tier, respectively. The European Processor Initiative (EPI) announced that a similar approach is under consideration for their upcoming processor architecture[52].

There are different ways of managing such a complex memory architecture. The high-performance memory tier may be used as a transparent cache, where hardware takes care of moving data between the tiers in a coherent manner. Alternatively, different (physical) address spaces may be defined for both tiers. In this case both placement of data objects as well as data movement between the tiers have to be managed in software. The KNL architectures support both.

Despite significant research efforts on the issue of leveraging such complex memory architectures[53], the exploitation for applications continues to be challenging. This is due to the lack of understanding on which memory tier needs to be chosen for placing data objects of real-life applications, e.g., the flagship applications in this project.

This led to developing the MFAT tool that allows to create a profile of the memory utilisation. The goal was to produce information on the lifetime of data objects and on the accession to these data objects. Assuming an architecture, where data placement and movement between different memory

[51] A. Sodani et al., "Knights Landing: Second Generation Intel Xeon Phi Product", IEEE Micro, 36, 2, 34-46, 2016 (doi:10.1109/MM.2016.25).

[52] F. Magugliani (EPI), "Framework Partnership Agreement in European Low-Power Microprocessor Technologies" 2021, https://www.european-processor-initiative.eu/wp-content/uploads/2021/08/EPI-Forum-Teratec-2021.pdf

[53] See, e.g.: K. Fatahalian et al., "Sequoia: Programming the memory hierarchy", SC'06 proceedings, 2006 (doi:10.1145/1188455.1188543); J. Park et al., "Location-aware cache management for many-core processors with deep cache hierarchy", SC'13 proceedings, 2013 (doi:10.1145/2503210.2503224); S. Christgau et al., "Leveraging a Heterogeneous Memory System for a Legacy Fortran Code: The Interplay of Storage Class Memory, DRAM and OS", MCHPC 2020 proceedings, 2020 (doi:10.1109/MCHPC51950.2020.00008).

tiers are managed by software, this information can be used to provide guidance for such data management decisions. The output of MFAT could in the future be used to place, e.g., frequently used (hot) data objects in a small high-performance memory tier implemented by HBM memory and place less hot data objects in a larger DDR-based memory tier.

4.2 Implementation

For MFAT we exploit a specific feature of the Linux kernel that is called "Idle Page Tracking"[54]. While a workload is being executed, one can track which of the allocated memory pages are being accessed and which remain idle. When enabled, this feature can be accessed through a simple application programming interface. A bitmap is made available through the virtual file system /sys where each bit corresponds to a memory page usage flag. These flags are only updated for user level memory pages, not for the memory used by the kernel internally. The usage flag of any such page is initially set to "idle", i.e., the page is marked as not (having been) used. The flag is changed to "active" whenever the kernel performs some operation on the page, e.g., doing I/O or resolving a page fault. It should be noted that in the kernels offered by typical Linux distributions, this feature is not enabled by default, i.e., MFAT requires the use of a specifically compiled kernel. As this limits the practical use of the tool, we consider MFAT at this point a proof-of-concept implementation.

One of the MFAT tools, which is called `mfat-mws`, directly reads and manipulates the kernel's data structures for managing the memory. Therefore it requires elevated privileges. When the application is started, the tool continuously traverses the active set of memory pages that are allocated by the application. For each page, the tool records the value of the usage flag. If the flag is set to "active", which indicates that the page has been accessed, then it is reset to "idle". In this way it is possible to track whether a given memory page has been accessed between two iterations during which the tool traversed the set of active memory pages.

A second MFAT component, a library called `mfat-memtrace`, is used to intercept calls of the application to the functions `malloc()` and `free()`, which are used to allocate or free dynamic memory. (Note that these functions are not necessarily called by the applications directly. For instance, for Fortran applications the compiler generates calls to these functions.) The library is dynamically linked by defining the environment variable `LD_PRELOAD`. This results in the aforementioned functions being replaced by wrapper functions that perform additional operations before calling the originally foreseen function. In this way, tracing information is collected about what data objects are allocated and when. Support for filtering the tracing information has been added to limit it to more relevant information, e.g., allocation of large memory regions.

In a post-processing step the information collected through the `mfat-mws` tool and the `mfat-memtrace` library can be combined. This allows correlating application data structures and the information collected by `mfat-mws`, which only contains the addresses of the active memory pages.

[54] Linux Foundation, "The Linux kernel user's and administrator's guide", https://www.kernel.org/doc/html/v5.15/admin-guide/mm/idle_page_tracking.html [accessed 2021-11-13].

4.2.1 Implementation details

The chosen approach for MFAT has some limitations and challenges that require particular consideration.

The major source of updates to the memory usage flags are page faults handled by the kernel. For accessing memory the processor's memory management unit (MMU) needs to translate virtual to physical addresses. To speed up such translations, the MMU has a (multi-level) translation lookaside buffer (TLB), which caches a limited number of maps. Whenever no mapping is known to the MMU, a page fault is generated and a fault handler in the kernel is invoked. These are usually minor page faults, i.e., the page is already present in memory, only the mapping in the MMU is missing. MFAT can only reliably determine the working set of applications that regularly cause minor page faults, i.e., the working set exceeds the capacity of all TLB levels. E.g., on an Intel(R) Xeon(R) CPU E7-4830 v4 there are 64 L1 data TLB entries and 1536 shared L2 TLB entries for 4KB data pages, which correspond to 6.25MB of data mappable via TLBs and a lower bound for the recognizable working memory set. Smaller workloads would be misdetected as mostly "idle" pages. This should not be a real limitation for typical HPC applications.

Another limitation concerns the use of huge pages, e.g., 2MB, 4MB, or 1GB pages, instead of the standard 4KB page. The use of huge pages is often beneficial for those HPC applications where the typical data objects are large. The reason is that less memory pages are required per data object. With huge pages, a larger volume of memory can be covered by this cache and fewer TLB updates have to be performed. Since there is only one memory usage flag per (huge) page, MFAT view on the memory use might become too coarse grained when, e.g., multiple data objects with different life-times and hotness are placed on the same page. While the TLBs for huge pages are smaller, they still map more memory and would require larger minimal working sets for analysis by MFAT.

While the approach used for MFAT can be used for multi-threaded applications, the collected data might in practice become difficult to interpret. It is currently not possible to correlate the collected data with specific threads. If the threads are only loosely synchronised then MFAT will show the behaviour that results from different threads executing different parts of the applications concurrently. In our evaluation, we restricted ourselves therefore to single-threaded execution of the application.

Thirdly, MFAT at this point assumes that the compute nodes that are being used do not support swapping of memory pages as a consequence of memory over-allocation. This assumption can be considered reasonable as compute nodes of large-scale HPC systems do not feature a swap device. It should, however, be noted that MFAT could be extended by tracking the information that shows whether a memory page is "unmapped".

Finally, MFAT assumes that all relevant data objects are dynamically allocated by the application itself, e.g., on the heap or stack. Any other memory pages, e.g., the mapping of the application binary or shared libraries, are ignored.

4.2.2 Security considerations

MFAT is based on the Linux kernel's "Idle Page Tracking" feature, which may raise security concerns. They are documented in the following without claiming sufficient competence for a full security risk assessment. Furthermore, we describe strategies on how to mitigate the identified risks.

Information about memory pages changing between "idle" and "active" state may provide side channel information that could possibly be exploited. Therefore the access to the memory page usage flag is restricted to processes with `CAP_SYS_ADMIN` capability.[55] Also the information about the mapping between virtual and physical addresses is restricted. Since the Linux kernel tracks the memory usage bits for the physical memory pages and MFAT needs to correlate this with the applications data objects, which are allocated at a virtual address, this mapping information needs to be available. In current Linux systems this information can be found in a virtual file called `/proc/$pid/pagemap`, where `$pid` stands for the numerical identifier of the considered process. Access to this virtual file also requires the `CAP_SYS_ADMIN` privilege.

To manage the necessary Linux capabilities, `mfat-mws` uses libcap, developed in the capabilities commands and library project. Using the command `setcap` the tool obtains the necessary privileges without having root privileges, e.g., through the setuid root mechanism.

`mfat-mws` can operate in two modes. Either it attaches to an already running process or it executes the workload as a child process which is thereafter being profiled. In the latter case, the Linux capabilities are dropped before the workload is executed. Only a single process will be profiled, any child processes created by the workload will be ignored. In both modes, `mfat-mws` will stop after the profiled process has terminated.

For future versions of MFAT we consider the implementation of a kernel module that would avoid the need for `mfat-mws` using Linux capabilities. Instead, it would run as an unprivileged user with access to only those information that are strictly required, i.e., the information about the usage of those memory pages that have been allocated by processes started by the same user as well as information about virtual and physical address mapping.

4.3 Experimental Results

In this section we collect a set of results obtained with the MFAT tools developed in this project.

In Fig. 47 we show how the working set size changes during a single time-step of the Fleur application. It shows that there are data objects with a footprint of about 600 MiByte that are actively used for a period of a few 100 s. Assuming a bandwidth to the large-capacity memory tier of about 100 GByte/s means that in the worst case, where all these data objects would have to be moved from the large-capacity to the high-performance memory tier and back, would require about 10 ms. The time for data movement is orders of magnitude smaller compared to the time during which the data is used.

[55] The concept of Linux capabilities was at some point introduced in the Linux kernel to manage privileges on a per-thread basis in a more fine-grained manner. Before, Linux only considered privileged and unprivileged processes.

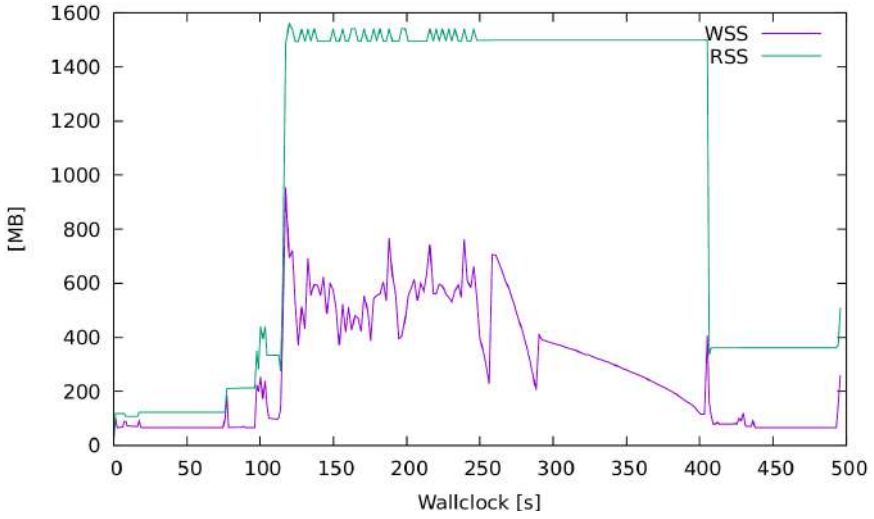

Fig. 47: The working set size (WSS) and real set set size (RSS) for a single-threaded Fleur time-step.

In Fig. 48 a different presentation of data produced by MFAT is shown for another application, namely Siesta. In this case bins of size 256 memory pages, each with a size of 4 kByte, are marked with a purple dot if at least one of the pages is idle during a given time interval but not actively used, or with a green dot if at least one of the pages is active. We observe that during this time step, there are about 1000 bins that during a period of a few seconds contain actively used pages. This corresponds to a memory footprint of about 1 MiByte. Different parts of the set of pages, that are actively used once during a time step, are accessed during such a time step and a different hotness could be associated with these bins. The most frequently used bins have a rather small footprint of about 300 kiByte and thus can for this particular workload be expected to fit in the processor cache.

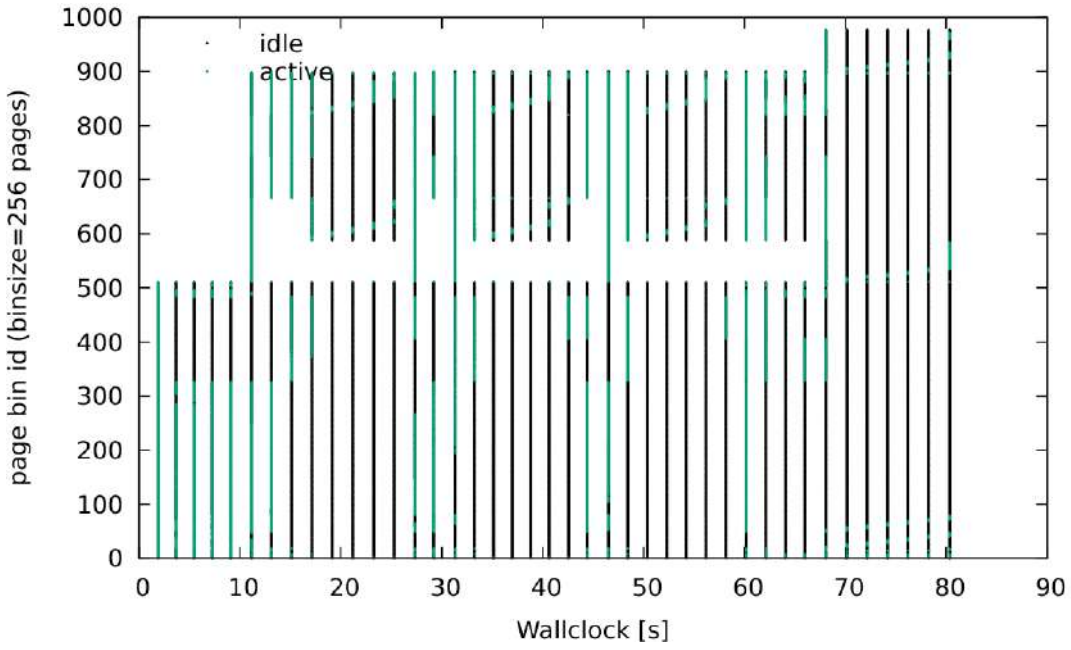

Fig. 48: Binning view of allocated memory pages for a single Siesta time step using a single thread.

5. Programming models

5.1. A proof-of-concept for GPU directive-based approaches

Motivation for a mixed approach

When the first hybrid systems with GPUs appeared on the HPC market, the only programming paradigm available for scientific computing was CUDA. This approach has the major disadvantage of binding the evolution of the code to a single specific technology provider, reducing the portability of the code in case new accelerators become available. Also, rewriting the code in CUDA requires a significant investment from the side of scientific communities. Over the years, many other approaches have appeared. Some are based on language extensions (e.g., OpenCL, Kokkos, CUDAFortran), while others are based on compiler directives (OpenACC and OpenMP). Nonetheless, none of them affirmed itself as a standard. NVIDIA pursued to sponsor CUDA (and minorly CUDAFortran) and OpenACC, while the support to the OpenMP5 directives for offloading has only recently gained relevance with the inclusion in OneAPI by Intel. More recently, accelerators by AMD claimed to be usable with OpenMP, even if HIP is the proprietary language. In any case, it is evident that none of these programming paradigms could be considered as universal and usable for all the devices. Also, it is quite difficult to decide if one of these could emerge as such in the near or medium future.

In this context, it is very difficult for a scientific community to decide in what direction to invest its efforts. Even though less effective, a popular choice, at least for the Fortran codes, is to move towards directive-based approaches (i.e., OpenMP5 and OpenACC). However, again, it is quite difficult to choose one single approach. In the present proof-of-concept we will explore the possibility of co-existence of OpenMP and OpenACC directives in a very simple case.

Implementation

Thanks to a fruitful collaboration with NVIDIA, the first approach to GPU porting of Quantum ESPRESSO and Yambo (and FLEUR, too) was based on the implementation with CUDAFortran. The advantage in this approach was the possibility to reduce the duplication of code with the usage of *cuf kernels.* Even though this approach has shown to be effective, CUDAFortran suffers from a minor support from NVIDIA, that is more focusing on OpenACC. Looking at the future, it seems likely that CUDAFortran will be only poorly supported, thus we are thinking about using other directive-based approaches, without revolutionizing the code. All this considered, we decided to add more backends for the GPU offloading. In doing this we wanted to preserve the modularization strategy adopted by the two codes, where some peculiar calls to libraries were wrapped and collected in Fortran modules or separated libraries developed by the same code developers, e.g., for the memory management. To this aim, a fundamental ingredient is that the different backends should preserve their interoperability.

We decided to make an intense use of preprocessor macros that activate the language chosen at compile time.

For this reason it was necessary to implement a simple application as a proof-of-concept called gpu-multi, based on many implementations of the SAXPY (Single-Precision A·X Plus Y) algorithm. In

this proof-of-concept we implemented different paradigms, all using directives to offload the SAXPY loop, including, e.g., CUDAFortran (with cuf kernel), OpenACC, and OpenMP5.

To better explain the adopted strategy, we report the SAXPY loop as implemented in the gpu-multi program:

```
subroutine saxpy_explicit(N, a, x, y)
 use kinds_m
 implicit none
  integer  :: N
  real(SP) :: a
  real(SP) DEV_ATTR :: x(N), y(N)
  integer i
  !
  !DEV_CUF kernel do
  !DEV_ACC data present(x,y)
  !DEV_ACC parallel loop
  !DEV_OMP5 target map(tofrom:x,y)
  !DEV_OMP5 parallel do
  !DEV_OMP  parallel do default(shared), private (i)
  do i = 1, N
   y(i) = y(i) + a * x(i)
  end do
  !DEV_OMP5 end target
  !DEV_ACC end data
  !
end subroutine saxpy_explicit
```

As it can be seen in the code above, just before the loop many directive lines are added. All these lines usually are treated as comments by the compiler, but with a proper definition of the macros (those with the prefix “DEV_”) it is possible to activate one or more of these lines as a directive for the compiler.

```
#if defined __CUDAF
#  undef  DEV_CUF
#  define DEV_CUF  $cuf
#elif defined __OPENACC
#  undef  DEV_ACC
#  define DEV_ACC  $acc
#elif defined __OPENMP5
#  undef  DEV_OMP5
#  define DEV_OMP5  $omp
#else
#  undef  DEV_OMP
#  define DEV_OMP  $omp
#endif
```

In the declaration it is possible to see the statement “DEV_ATTR”: this, thanks to a macro, can be substituted with the proper statement for the language chosen. In particular this is necessary for the CUDAFortran language that needs to declare as “device” a variable allocated in the device memory.

Speaking of memory, it is also relevant to note that in the OpenACC and OpenMP5 directives it is necessary to specify that the variables used in the loop are already initialized in the device memory. This was necessary in order to preserve the modular structure of the codes, as explained above.

While this simple example shows that it is possible to operate at the same level with different directive-based approaches, for more complex codes there are many difficulties and/or limitations. It should be noted that the macro only replaces the sentinel in the directives: the clauses should be explicitly managed and in many cases should be tracked down differently for the different programming models (in some cases, the memory management can be different in OpenMP5 and OpenACC). For example, in CUDAFortran, there is the need to explicitly allocate “device variables”, which are automatically created by OpenACC and OpenMP. Also, the porting when limited only to a directive-based layer cannot ensure that the maximum level of efficiency is achieved and in some cases it would be desirable to have an explicit backend in CUDA or HIP (or other low-level language) to shrink the performances. Nonetheless, this small proof-of-concept is useful to understand that we have, at this level, a quite easy way to convert codes which are already ported in a directive-based model to a different one.

CINECA GPU Hackathon

During the CINECA GPU Hackathon (organized by CINECA in collaboration with the GPU Hackathon program[56]) we had the opportunity to discuss with our tutors about our proof-of-concept (gpu-multi) and about its implementation in Yambo. By the end of the hackathon we were able to obtain a version of our library deviceXlib ported on OpenACC and to maintain the other backend with CUDAFortran using the strategy explained above. As already explained, Yambo is already ported to the GPU using CUDAFortran. Moreover, we started to work on Yambo integrating the new version of deviceXlib as an external library and porting some drivers with the double backend. Now we are able to perform a Hartree-Fock calculation using Yambo compiled with the OpenACC backend.

5.2. HPX programming model

HPX (High Performance ParalleX) is a general purpose C++ runtime system for parallel and distributed applications of any scale. It strives to provide a unified programming model which transparently utilizes the available resources to achieve unprecedented levels of scalability. This library strictly adheres to the C++11 Standard and leverages the Boost C++ Libraries which makes HPX easy to use, highly optimized, and very portable. HPX is developed for conventional architectures including Linux-based systems, Windows, Mac, and the BlueGene/Q, as well as accelerators such as the Xeon Phi.

During the course of the project we investigated the possibility of using HPX in scientific applications. We decided to study HPX in a mini-app benchmark that performs tall-and-skinny general matrix multiplication (TSGEMM). The data layouts for this benchmark are identical to the plane-wave DFT

[56] https://www.gpuhackathons.org/

codes (tall-and-skinny matrices are wave-functions distributed in slabs beetweek MPI ranks, final matrix is a subspace Hamiltonian or overlap matrix distributed in 2D block-cyclic ScaLAPACK fashion).

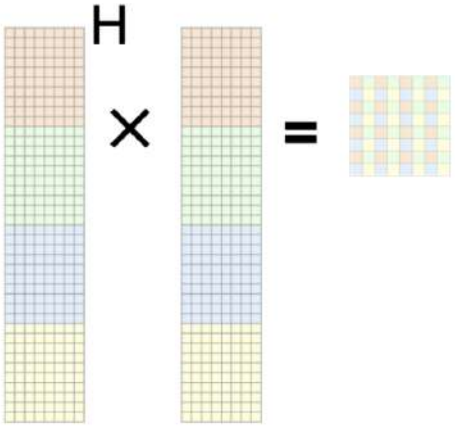

Fig. 49: Tall-and-skinny matrix multiplication problem. Initial and final matrix distributions are different and correspond to the data layout in the actual DFT codes.

In the first part of the project we focused on the custom implementation of the TSGEMM with a lot of attention paid to the MPI data exchange. The details of the work are available in deliverable D4.5[57]. In the final part of the project we focused on two action points: 1) clean implementation of TSGEMM benchmark using the tiling mechanism of DLA-Future library, and 2) possibility to overlap two consecutive computations (matrix multiplication of our benchmark and Cholesky factorisation of the resulting matrix) using the benefit of the HPX task-based approach. DLA-Future is a HPX-based library developed within the PRACE-6IP project with the aim of delivering the scalable eigen-value solver[58]. The tiling mechanism of DLA-F provides all the necessary building blocks for matrix operations and has shown to be very useful for the TSGEMM benchmark. The final results of the benchmark are presented in the figures below and the discussion of the results follows.

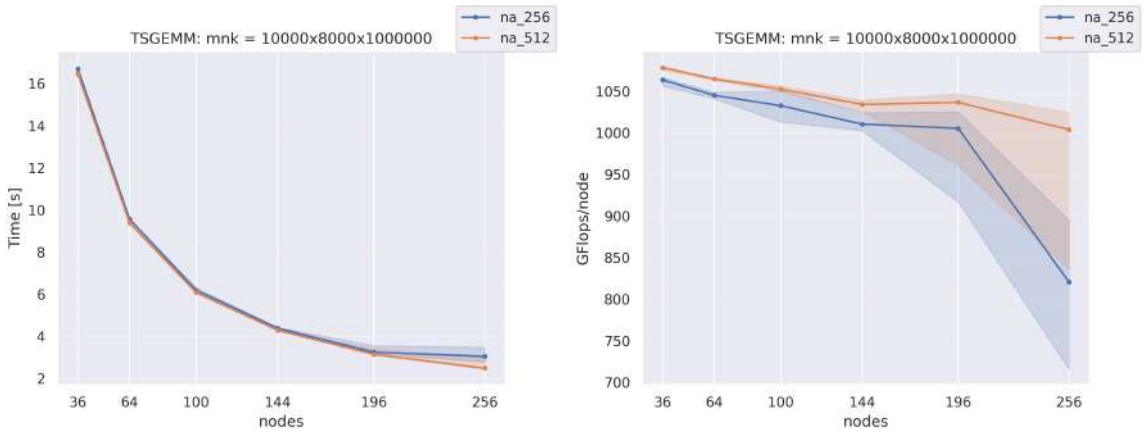

Fig. 50: Tall-and-skinny matrix multiplication benchmark. The dimensions of the matrix multiplication problem [m,n,k] are [10'000, 8'000, 1'000'000]. The benchmark was executed on 36-256 dual socket multi-core nodes of Piz Daint. Each node is equipped with two 18-core Intel Broadwell processors @2.1 Ghz. Peak performance of the node is ~1.2 TFlop/s. Left: execution time in seconds. Right: performance per node, GFlop/s. The internal parameter `na` corresponds to the DLAF-F tile size.

57 http://www.max-centre.eu/sites/default/files/D4.5_Final%20report%20on%20codes%20profiling%20and%20bottlenck%20identification.pdf

58 https://github.com/eth-cscs/DLA-Future

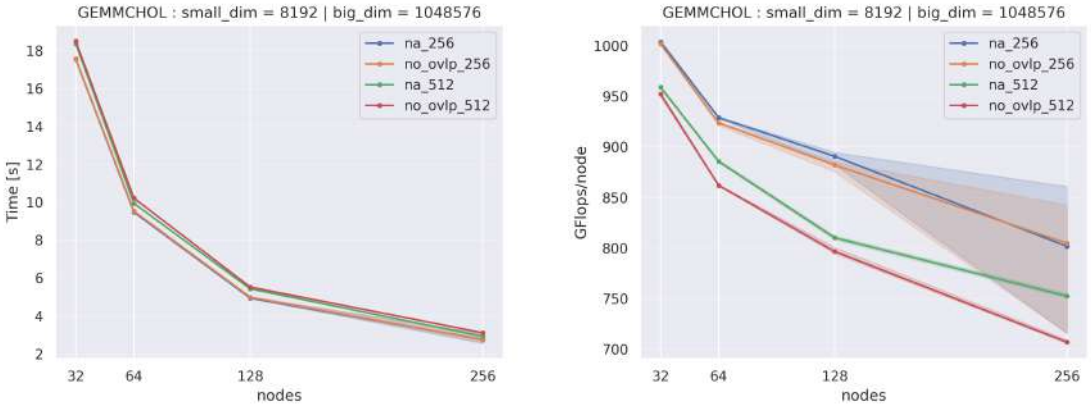

Fig. 51: Tall-and-skinny matrix multiplication overlapped with Cholesky factorisation. The dimensions of the matrix multiplication problem [m,n,k] are [8'192, 8'192, 1'048'576]. The benchmark was executed on 36-256 dual socket multi-core nodes of Piz Daint. Each node is equipped with two 18-core Intel Broadwell processors @2.1 Ghz. Peak performance of the node is ~1.2 TFlop/s. Left: execution time in seconds. Right: performance per node, GFlop/s. The internal parameter `na` corresponds to the DLAF-F tile size. `No overlap` means that TSGEMM was executed first, followed by the global MPI_Barrier followed by Cholesky factorisation. 'Overlap` means that both TSGEMM and Cholesky factorisation were expressed as a single graph of dependent tasks. Once the tile of the final matrix is computed its factorisation can be started immediately. The difference between performance of two executions is small because TSGEMM is a much more 'heavy' task that consumes most of the run-time.

To summarise the activities of exploring the HPX run-time system: HPX represents a modern task-based C++ library for node-level and shared memory parallelisation. It permits to express the computational work in terms of a DAG of tasks that can be executed concurrently on the multi-core and hybrid nodes. Most importantly, HPX can co-exist with the other programming models, like OpenMP.

5.3. Libconv

The auto-tuned *libconv* convolution library has been proven to be a useful and powerful addition to BigDFT. Its integration is challenging, as a lot of convolutions calls are already scattered in BigDFT, and the new convolutions API introduced with *libconv* require to change in several places at the computation kernel level. Memory requirements for vectorization are also important, as properly aligned memory must be used. This can be done within the *futile*[59] library, but must be handled with care.

In order to keep intrusivity in the existing code minimal, and to provide *libconv* as an option instead of a complete replacement of existing convolutions, a new interface library is being developed. *Liborb* (library for orbital manipulations) aims at handling in a single place all the orbital-related tasks. The aim of this library is to provide an API for the flexible definition of self-consistent field (SCF) operations at a high level. The goal is to separate the implementation of the orbitals representation

[59] Library handling most common FORTRAN low-level operations, like memory management, profiling routines, I/O operations, such as yaml output and parsing for fortran programs. It also provides wrappers routines to MPI and linear algebra operations: https://l_sim.gitlab.io/futile/

from the SCF algorithms using it, so that the user is kept isolated from the basis sets employed for the discretization.

Liborb handles the specification of the orbital, which can be discretized in wavelet bases, gaussian bases, or real space. The centralization of these representations simplifies the wrapping of all calls to *libconv*, which can be used mostly for the wavelet basis. *Liborb* indeed handles the scalar products between sets of orbitals, the communication of these orbitals between processes, and the application of Hamiltonian operators on these orbitals, and I/O for these bases.

Conversion between all basis representations is also possible within the library, and is useful in most of the current developments of BigDFT.

The wrapping of all *libconv* operations in a single point will make the performance evaluation much easier in order when the selection of the most efficient flavour of convolutions is possible.

Libconv itself has also been improved, with the support of SVE instructions within the *BOAST*[60] framework, to generate automatically SVE-vectorized convolutions, making the best possible use of modern ARM processors, such as the A64FX used in the Fugaku supercomputer.

	Benchmark time (ms)	
OpenMP Threads	**1**	**48**
Ref (Fortran, no OpenMP) + GCC autovec	52961	52961
libconv – no vectorization + GCC autovec	17525	549
libconv – NEON vectorization + GCC autovec	15930	511
libconv – SVE vectorization + GCC no autovec	13604	455
libconv – SVE vectorization + GCC autovec	13689	456

Table 12: Preliminary timings for 300 full applications of a « MagicFilter » convolution, double precision with sizes 124*132*130 on an A64FX processor. Compiler used is GCC10. Ref is just for results checkings and not the current BigDFT implementation. We can see that the generation of SVE instructions directly from the libconv library using BOAST outperforms the auto vectorized versions of the same convolutions and the NEON version, while the compiler cannot further optimize the resulting convolutions through vectorization. All tested libconv-generated convolutions are using unrolling/dimensions reordering and other optimizations automatically and generated in C, with wrappers created for C and Fortran interfacing. OpenMP scaling over 48 cores is ~30 for each variant, showing that the scaling should not be affected by the extended use of SVE instructions. Flags for neon auto vectorization were -O3 -ftree-vectorize -march=armv8-a+sve -msve-vector-bits=512 .

[60] Metaprogramming framework to produce portable and efficient computing kernels for HPC applications: https://doi.org/10.1177/1094342017718068 and https://github.com/Nanosim-LIG/boast

6. Conclusions

In the first part of this deliverable we performed an in-depth analysis of the MaX flagship codes on several CPU microarchitectures, to characterize their performance in relation to the hardware. The outcome is meant to help the improvement of the codes and the assessment of the work performed during the MaX project (by comparing the results on different versions of the applications). Importantly, it provides data of relevance to the hardware and system design, to support trade-off decisions when delivering new products on the HPC market. Of particular relevance is our analysis in terms of energy-to-solution, since this will be a major constraint for the design of future HPC systems.

Thanks to the partnership with ARM, we explored the performance of two MaX applications on the ARM Neoverse N1 chip, which is the precursor of the Nvidia GRACE cpu of relevance to the Sipearl Rhea processor, as part of the strategy of the EPI.

In the third section, we reported about the work on a tool for the analysis of the memory usage of the application, which is a very delicate issue when looking at the next generation of supercomputers, where the memory hierarchies (from HBM to SSD) have a particular importance.

In the last part of the deliverable, we reported some proof-of-concept results related to the software design of the MaX applications. On the one hand, we continued the exploration of programming paradigms and their evaluation with respect to code portability. On the other hand, in the perspective of the 'separation of concerns' concept, we worked on domain-specific libraries to deal with optimization with respect to hardware features such as vectorization and asynchronous models.

All the work performed and reported here has taken benefit from the tight collaboration with the code developers, influencing and getting feedback from other MaX work-packages (in particular WP1, WP2, and WP3) activities. The close interaction among technologists from HPC computing centres, scientists, and code developers has continuously fueled the progress of the work that we have presented: a concrete application of the co-design cycle that we proposed at the beginning of MaX in 2018.

Abbreviations

AIMD	Ab-Initio Molecular Dynamics
API	Application Programming Interface
BPU	Branch Prediction Unit
CoE	Centre of Excellence
CPU	Central Processing Unit
DAG	Direct Acyclic Graph
DFT	Density Functional Theory
DRAM	Dynamic Random Access Memory
EPI	European Processor Initiative
FLOP	Floating-Point Operations per Second
FLOPs/W	FLOPs per Watt
FMA	Floating Point Multiply-Accumulate Operation
FP	Floating Point
GCC	GNU Compiler Collection
GFLOPs	Giga FLOPs
GGA	Generalized Gradient Approximation
HBM	High-Bandwidth Memory
HPC	High Performance Computing
HPX	High Performance ParalleX
HW	Hardware
IPC	Instructions Per Cycle
KNL	Knights Landing
LDA	Local Density Approximation
MC	Management Controller
MCDRAM	Multi-Channel DRAM
MFAT	Multi-Fastener Analysis Tool
MMU	Memory Management Unit
MOP	Macro Operation
µOP	Micro Operations
MPI	Message Passing Interface

OCC	On-Chip Controller
PAW	Projector Augmented Wave
PBE	Perdew-Burke-Ernzerhof
PMC	Performance Monitoring Counter
PMPI	Profiling Message Passing Interface
PoC	Proof-of-Concept
QE	Quantum ESPRESSO
RAM	Random Access Memory
RAPL	Running Average Power Limit
SCF	Self-Consistent Field
SMT	Simultaneous Multithreading
SoC	System-on-Chip
SSD	Solid-State Drive
SVE	Scalable Vector Extension
SW	Software
TDP	Thermal Design Power
TFLOPs	Tera FLOPs
TLB	Translation Lookaside Buffer
TMAM	Top-down Microarchitecture Analysis Method
TSGEMM	Tall-and-Skinny General Matrix Multiplication