Performance study of HPC applications on an Arm-based cluster using a generic efficiency model

Fabio Banchelli *, Kilian Peiro †, Andrea Querol ‡, Guillem Ramirez-Gargallo §,
Guillem Ramirez-Miranda ¶, Joan Vinyals ‖, Pablo Vizcaino **, Marta Garcia-Gasulla ††, Filippo Mantovani ‡‡
Barcelona Supercomputing Center
Email: *fabio.banchelli@bsc.es, †kilian.peiro@bsc.es, ‡andrea.querol@bsc.es, §guillem.ramirez@bsc.es,
¶guillem.ramirez.miranda@bsc.es, ‖joan.vinyals@bsc.es, **pablo.vizcaino@bsc.es,
††marta.garcia@bsc.es, ‡‡filippo.mantovani@bsc.es

***Abstract*—HPC systems and parallel applications are increasing their complexity. Therefore the possibility of easily study and project at large scale the performance of scientific applications is of paramount importance. In this paper we describe a performance analysis method and we apply it to four complex HPC applications. We perform our study on a pre-production HPC system powered by the latest Arm-based CPUs for HPC, the Marvell ThunderX2. For each application we spot inefficiencies and factors that limit their scalability. The results show that in several cases the bottlenecks do not come from the hardware but from the way applications are programmed or the way the system software is configured.**

***Index Terms*—Performance analysis, High Performance Computing, Parallel Applications, Arm, ThunderX2**

I. Introduction

With the end of Moore's law both hardware and software of large High-Performance Computing (HPC) systems are becoming more complex. They tend to include more and more specific units accelerating certain workloads orchestrated by frameworks/programming models that should help the programmer. In the panorama of HPC architectures emerging in the recent years there is the Arm architecture. In Nov 2018, for the first time, an Arm-based system, Astra, has been ranked in the Top500. Astra is powered by Marvell ThunderX2 CPUs.

As counterpart of this mosaic of architectures we have complex scientific applications running on HPC clusters. The domain scientist developing them should focus on the science not worrying too much about the underlying hardware. So tools easing the understanding of the performance bottlenecks of complex parallel applications are needed.

In this paper we leverage a performance analysis and modeling method for an in depth study of four parallel applications on an HPC cluster powered by the same CPUs as the Astra supercomputer. For validating our tests we selected a set of complex HPC applications proposed as challenge within the Student Cluster Competition held at the ISC conference [1].

The main contributions of this paper are: *i)* we describe a performance analysis method that can be applied to parallel applications; *ii)* we apply the method to four real HPC applications used in real scientific use-cases; *iii)* we study the outcome of performance analysis data gathered on a state-of-the-art HPC cluster based on emerging technology Arm CPUs.

The remaining part of the document is structured as follows: in Section II we report about similar works and how we contribute to the state-of-the-art. Section III introduces the experimental setup as well the HPC applications studied in the rest of the paper. In Section IV we summarize the methodology and metrics used in the performance analysis. Sections V, VI, VII, and VIII are dedicated to the single-node and multi-node analysis of each of the applications. We close the paper with our comments and considerations in Section IX.

II. Related Work

A review of performance analysis tools is provided by Niethammer et al. in the book [2]. Burtscher et al. in [3] present a tool for performance diagnosis of HPC applications. They focus on analyzing architectural features while we leverage portable global metrics for measuring the efficiency of HPC applications. Stanisic et al. in [4] conduct a performance analysis studies of HPC applications, but focusing on mobile Arm-based CPU. Garcia-Gasulla et al. in [5] also evaluate a CFD code on an Arm-based cluster however their focus is on portability of performance across multiple architectures. We complement this work providing an evaluation with a larger set of HPC applications and we provide an analysis method that is mostly architecture independent. Calore et al. in [6] use an approach similar to ours, but they apply it only to a single CFD code and on a previous generation of server-grade Arm chips. McIntosh-Smith et al. in [7] focus on comparing the performance of different applications on Arm-based and Intel-based architecture. We extend this work including an in depth study of performance bottlenecks and scalability limitations on the same Arm-based CPU studied by them.

III. Experimental Setup

A. The Dibona Cluster

The Dibona cluster has been integrated by Bull/ATOS within the framework of the European project Mont-Blanc 3 [8]. It integrates 40 Arm-based compute nodes powered by two Marvell ThunderX2 CN9980 processors[1], each with 32 Armv8 cores at 2.0 GHz, 32 MB L3 cache and 8

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

 DOI 10.1109/PDP50117.2020.00032

	Pennant	CP2K	OpenFOAM	Grid
URL	github.com/lanl/PENNANT	github.com/cp2k	openfoam.com	github.com/paboyle/Grid
Scientific domain	Lagrangian hydrodynamics	Quantum chemistry / solid state physics	CFD	Lattice QCD
Version / Commit	v0.9	v6.1.0	v1812	251b904
Programming language	C++	F77 & F90	C++	C++
Lines of code	3.5k	860k	835k	75k
Parallelization scheme	MPI	MPI	MPI	MPI
Compiler	Arm/19.1	GNU/8.2.0	Arm/19.1	Arm/19.1
Scientific libraries	-	BLAS, LAPACK, FFT via Armpl/19.1	Armpl/19.1	Armpl/19.1
Focus of analysis	4 steps	4 steps QMMM	4 steps SIMPLE solver	20 steps + 1 eval
Input set	Leblancx4	wdftb-1 (FIST, QuickStep DFTB, QMMM)	DrivAer model	Lanczos algorithm

TABLE I: Summary of HPC applications and their configurations.

DDR4-2666 memory channels. The total amount of RAM installed is 256 GB per compute node.

Compute nodes are interconnected with a fat-tree network, implemented with Mellanox IB EDR-100 switches. A separated 1 GbE network is employed for the management of the cluster and a network file system (NFS).

Dibona runs Red Hat Enterprise Linux Server release 7.5 with kernel v4.14.0 and it uses SLURM 17.02.11 as job scheduler. The Arm system software ecosystem also includes an Arm compiler based on LLVM and a set of optimized arithmetic libraries called Arm Performance Libraries (Armpl). Both are distributed in a single package that we leverage for some of the applications. See Table I for more details.

B. The HPC applications

For our study we use four scientific parallel applications: Pennant, OpenFOAM, CP2K, and Grid. Table I summarizes relevant details about each application. We want to stress the fact that in the current work we evaluate a set of applications that are used in a daily basis either for procurements of large HPC systems (e.g., the Pennant mini-app) or for performing cutting edge research in academia and industry (e.g., OpenFOAM, CP2K, and Grid). The selection of the applications has been inspired by the challenges proposed to the teams participating in the Student Cluster Competition held at ISC Conference (Frankfurt) in 2018 and 2019. We consider the application selection representative of different HPC workloads and heterogeneous scientific fields.

IV. Methodology for Performance Analysis

In this paper we leverage the metrics for modelling the performance of parallel applications developed and promoted by the European Center of Excellence Performance Optimization and Productivity[2] (POP) defined in [9]. For this reason we often call the efficiency metrics used for our performance analysis *POP metrics*. These metrics determine the efficiency of the MPI parallelization and can be computed for any MPI application. While they are objective, they are not conclusive: POP metrics represent indicators that guide a following detailed analysis to spot the exact factors limiting the scalability.

In this work we focus only on analyzing the parallel efficiency of the MPI parallelization. Also, all the analysis are performed on traces obtained from real runs. A trace contains information about all the processes involved in an execution, and they can include, among others, information on MPI or I/O activity as well as hardware counters.

We followed the same steps to analyze each application:

- Run the application with a relevant input and core count and obtain a trace from this run.

- Based on this trace, determine the focus of the analysis. This is step is performed disregarding the initialization and finalization phases, identifying the iterative pattern (if possible) and selecting some representative iterations.

- Compute the performance metrics for different number of MPI processes in a single node, in our case from 1 to 64. Based on these metrics we analyze in detail the main limiting factors and performance issues when scaling inside a node.

- Compute the performance metrics when using multiple nodes, in our case from 1 to 16 nodes (corresponding to 64 to 1024 MPI processes). Note that for the multi node study we used always full nodes (i.e., 64 MPI processes per node). Guided by the results on the metrics determine the issues that limit the scalability of the code on multiple nodes.

All the studies in this paper are done with strong scalability, but the methodology and metrics can be applied also to weak scaling codes.

A. Metrics for Performance Analysis

For the definition of the POP metrics needed in the rest of the paper we use the simplified model of execution depicted in Figure 1.

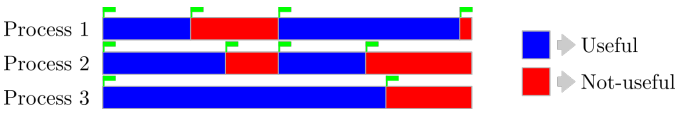

Fig. 1: Example of parallel execution of 3 MPI processes.

We call $P = \{p_1, \ldots, p_n\}$ the set of MPI processes. Also we assume that each process can only assume two states over the execution time: the state in which it is performing computation, called *useful* (blue) and the state in which it is not performing computation, e.g., communicating to other processes, called *not-useful* (red). For each MPI process p we can therefore define the set $U_p = \{u_1^p, u_2^p, \ldots, u_{|U|}^p\}$ of the time intervals where the application is performing useful

[2] https://pop-coe.eu

computation (the set of the blue intervals). We define the sum of the durations of all useful time intervals in a process p as:

$$D_{U_p} = \sum_{U_p} \blacksquare = \sum_{j=1}^{|U_p|} u_j^p \quad . \tag{1}$$

Similarly we can define $\overline{U}_p$ and $D_{\overline{U}_p}$ for the red intervals.

The *Load Balance* measures the efficiency loss due to different load (useful computation) for each process. Thus it represents a meaningful metric to characterize the performance of a parallel code. We define L_n the *Load Balance* among n MPI processes as:

$$L_n = \frac{\text{avg}_{|P|}\left(\sum_{U_p} \blacksquare\right)}{\max_{|P|}\left(\sum_{U_p} \blacksquare\right)} = \frac{\sum_{i=1}^{n} D_{U_i}}{n \cdot \max_{i=1}^{n} D_{U_i}}$$

The *Transfer efficiency* measures inefficiencies due to data transfer. In order to compute the *Transfer efficiency* we need to compare a trace of a real execution with the same trace processed by a simulator assuming all communication has been performed on an ideal network (i.e., with zero latency and infinite bandwidth). We define T_n the *Transfer efficiency* among n MPI processes as:

$$T_n = \frac{\max_{|P|} t'_p}{\max_{|P|} t_p}$$

Where t_p is the real runtime of the process p, while t'_p is the runtime of the process p when running on an ideal network.

The *Serialization Efficiency* measures inefficiency due to idle time within communications (i.e., time where no data is transferred) and is expressed as:

$$S_n = \frac{\max_{|P|} D_{U_p}}{\max_{|P|} t'_p}$$

Where D_{U_p} is computed as in Eq. 1 and t'_p is the runtime of the process p when running on an ideal network.

The *Communication Efficiency* is the product of the *Transfer efficiency* and the *Serialization efficiency*: $C_n = T_n \cdot S_n$. Combining the *Load Balance* and the *Communication efficiency* we obtain the *Parallel efficiency* for a run with n MPI processes: $P_n = L_n \cdot C_n$. Its value reveals the inefficiency in splitting computation over processes and then communicating data between processes. A good value of P_n *i)* ensures an even distribution of computational work across processes and *ii)* minimizes the time spent in communicating data among processes. Once defined the *Parallel efficiency*, the remaining possible source of inefficiencies can only come from the blue part of Figure 1, so from the useful computation performed within the parallel applications when changing the number of MPI processes. We call this *Computation Efficiency* and we define it in case of strong scalability as:

$$U_n = \frac{\sum_{P_0} D_{U_p}}{\sum_{P} D_{U_p}}$$

where $\sum_{P_0} D_{U_p}$ is the sum of all useful time intervals of all processes when running with P_0 MPI processes and $\sum_{P} D_{U_p}$ is the sum of all useful time intervals of all processes when running with P MPI processes, with $P_0 < P$. The possible cause of a poor Computation Efficiency can be investigated using metrics derived from processor hardware counters (e.g., number of instructions, number of clock cycles, frequency, Instructions Per Clock-cycle (IPC)).

Finally we can combine the efficiency metrics introduced so far in the *Global Efficiency* defined as $G_n = P_n \cdot U_n$.

B. Performance Tools

For the analysis performed in this paper we adopt a set of tools developed within the Barcelona Supercomputing Center.

Extrae [10] is a tracing tool developed at BSC. It collects information such as PAPI counters, MPI and OpenMP calls during the execution of an application. The extracted data is stored in a trace that can be visualized with Paraver.

Paraver [11] takes traces generated with Extrae and provides a visual interface to analyze them. The traces can be displayed as timelines of the execution but the tool also allows us to do more complex statistical analysis.

Clustering is a tool that, based in a Extrae trace, can identify regions of the trace with the same computational behavior. This classification is done based on hardware counters defined by the user. Clustering is useful for detecting iterative patterns and regions of code that appear several times during the execution.

Tracking helps to visualize the evolution of the data clusters across multiple runs when using different amount of resources and exploiting different levels of parallelism.

Combining the metrics defined in Section IV-A and the tools just introduced, we are able to summarize the *health* of a parallel applications running on a different number of MPI processes in a heat-map table. Rows indicate different POP metrics while columns indicate the number of MPI processes of a given run. Each cell is color-coded in a scale from green, for values close to 100% efficiency; yellow, for values between 80% and 100% of the efficiency; down to red, for values below 80% efficiency.

V. Pennant

Pennant is a mini-app that implements unstructured mesh physics using Lagrangian staggered-grid hydrodynamics algorithms. It is written in C++ by Los Alamos National Laboratory [12]. We study the MPI-only version of Pennant. There is also an optional hybrid implementation with MPI+OpenMP which we did not study. We used the Arm HPC compiler for our studies. The application does not require additional libraries. The input set used in our study is Leblancx4. It is a multi-node version of the Leblanc problem which contains 3.69×10^6 quad zones within a rectangular mesh.

A. Single-node performance analysis

Figure 2 shows the efficiency metrics introduced in Section IV-A for Pennant varying the number of MPI processes

within a compute node. Since all values are above 90% we can conclude that the parallel performance of Pennant within one node is good. The only varying factor is the IPC scalability that drops from 1.17 to 1.11 (-7%) when using 64 MPI processes. This behavior is expected when several processes are concurrently using all the resources of a compute node.

	2	4	8	16	32	64
Global efficiency	99.59	100.00	99.19	99.60	98.53	93.10
├ Parallel efficiency	99.59	99.80	99.20	99.16	99.00	98.02
├ Load balance	99.94	99.90	99.26	99.27	99.52	99.19
└ Communication eff.	99.65	99.90	99.93	99.89	99.47	98.82
├ Serialization eff.	99.67	99.92	99.99	100.00	99.71	99.02
└ Transfer eff.	99.98	99.98	99.94	99.89	99.76	99.80
└ Computation scalability	100.00	100.20	100.00	100.44	99.53	94.98
├ IPC scalability	100.00	100.01	99.96	100.32	99.39	94.82
├ Instruction scalability	100.00	100.08	100.03	100.11	100.03	100.00
└ Frequency scalability	100.00	100.11	100.01	100.01	100.11	100.16

100% 95% 90% 85% <80%

Fig. 2: Single-node efficiency metrics of Pennant

B. Multi-node performance analysis

Figure 3 shows the efficiency metrics of Pennant when scaling beyond a single node. As with single-node performance, Pennant presents a good scalability behavior up to 16 nodes. However, projecting the trend to a higher number of nodes, we can expect that the most limiting factors for a good scalability will be *Transfer* and *Serialization efficiency*.

	64	128	256	512	1024
Global efficiency	97.84	94.06	95.68	93.41	92.57
├ Parallel efficiency	97.84	94.24	95.47	94.58	93.06
├ Load balance	99.15	98.09	98.63	98.17	98.80
└ Communication eff.	98.67	96.08	96.80	96.34	94.19
├ Serialization eff.	98.88	96.31	97.65	98.08	97.73
└ Transfer eff.	99.80	99.76	99.13	98.22	96.38
└ Computation scalability	100.00	99.81	100.22	98.76	99.47
├ IPC scalability	100.00	99.91	100.40	99.31	100.62
├ Instruction scalability	100.00	100.04	99.82	99.37	98.69
└ Frequency scalability	100.00	99.86	100.00	100.07	100.17

100% 95% 90% 85% <80%

Fig. 3: Multiple-node efficiency metrics of Pennant

To find the root cause of *Transfer*, we studied the behavior of MPI calls during the iterations of Pennant. We observed that `Allreduce` calls present an irregular exit pattern in some cases. This phenomenon can be seen in Figure 4 where we show a Paraver timeline of Pennant running on 1024 MPI processes. In this view the x-axis represents time (window of 300 μs) while the y-axis shows one MPI process per line (256 processes). In this case the color represents the MPI call executed by a process. Pink corresponds to `Allreduce` and white depicts the time spent by the processes running non-MPI code (also known as *useful computation*).

The unaligned exit of the `Allreduce` call is due neither to load imbalance (all the processes have reached the collective call) nor to preemptions (we verified in the trace that the frequency of the processes does not change). We can conclude that late-exit of some processes is due to the MPI implementation deployed in Dibona.

We also observe in some iterations a load imbalance that is produced by a preemption of one or more processes. This causes a delay in some `Waitall` calls that are waiting for

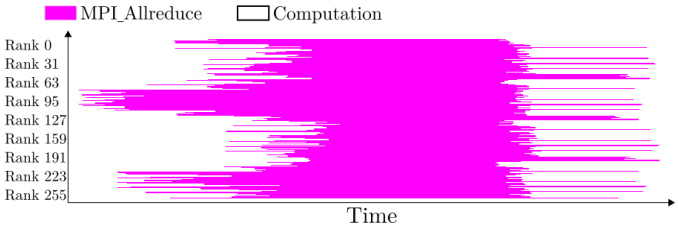

Fig. 4: Timeline of `Allreduce` of Pennant.

a message from the preempted process and is affecting the *Serialization efficiency*. We consider that these preemptions are due to some processes from the system software (also known as system noise). We can say that, although Pennant has a good scaling in multiple nodes, it is sensible to system noise and MPI implementation misbehavior.

VI. OpenFOAM

OpenFOAM is an open-source Computational Fluid Dynamics (CFD) software. There are different branches of the OpenFOAM project. We used OpenFOAM-v1812, maintained by the ESI-OpenCFD company combined with the DrivAer model[3] for aerodynamic studies of a car.

This model requires the executions of different binaries corresponding to different phases of the simulation (e.g., decomposePar, renumberMesh, checkMesh, potentialFoam and simpleFoam). We focused our analysis on the simpleFoam solver[4], which uses the SIMPLE (Semi-Implicit Method for Pressure Linked Equations) algorithm for incompressible, turbulent flow.

A. Single-node performance analysis

Figure 5 shows the efficiency metrics of OpenFOAM when executing in a single node.

	2	4	8	16	32	64
Global efficiency	99.10	92.33	87.34	79.75	58.02	58.22
├ Parallel efficiency	99.10	95.18	94.94	92.11	89.31	85.32
├ Load balance	99.80	97.37	96.04	94.20	94.84	91.60
└ Communication eff.	99.30	97.75	98.86	97.78	94.17	93.14
├ Serialization eff.	99.34	97.86	99.14	98.62	95.38	95.60
└ Transfer eff.	99.95	99.88	99.72	99.15	98.73	97.42
└ Computation scalability	100.00	97.01	92.00	86.58	64.97	68.24
├ IPC scalability	100.00	99.11	95.83	88.41	64.73	69.34
├ Instruction scalability	100.00	97.80	95.91	97.86	100.12	98.46
└ Frequency scalability	100.00	100.09	100.09	100.07	100.24	99.95

100% 95% 90% 85% <80%

Fig. 5: Single-node efficiency metrics of OpenFOAM.

The most limiting factor for the *Global efficiency* is the IPC scalability. We verified that the number of clock cycles increases with the number of processes while the number of useful instructions remains constant. This means that each instruction needs more clock cycles to finish, this effect is usually caused by the saturation of some of the hardware resources within the compute node. The memory bandwidth tends to be the most scarce resource when using all cores

[3] http://www.aer.mw.tum.de/en/research-groups/automotive/drivaer/
[4] http://openfoamwiki.net/index.php/SimpleFoam

in a single node. To verify if the saturation of the memory bandwidth is affecting the IPC we could check the L3 cache misses counters, however this counter is not accessible in Dibona. While we cannot confirm our hypothesis with a direct measurement due to the impossibility of accessing the L3 cache misses counter, we can discard other causes for the IPC drop looking at other hardware counters.

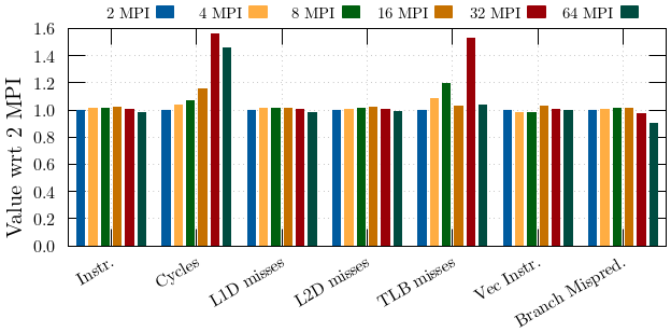

Fig. 6: Hardware counters in OpenFOAM single-node.

In Figure 6 we report the cumulative value of different hardware counters with respect to the values recorded when executing with 2 MPI processes. We notice that none of the hardware counters have variation in the bar corresponding to 64 MPI processes, confirming that none of the hardware resources monitored with those hardware counters are the cause of the IPC drop. Another indirect confirmation of the drop of IPC due to the lack of memory bandwidth come from the cross check of the IPC values when running 32 MPI processes pinned on one socket (IPC = 0.58) versus 16 MPI processes per socket (IPC = 0.80). As a side note, we can point out that the number of TLB misses can explain the lower IPC on 32 MPI processes compare to the 64 MPI processes.

We conclude that OpenFOAM saturates the memory bandwidth when running 16 MPI processes per socket.

B. Multi-node performance analysis

Figure 7 shows the efficiency metrics measured when running OpenFOAM on multiple nodes. Several factors are compromising the efficiency: *Load Balance*, *Serialization*, *Transfer* and *Instruction scalability*. The main limiting factor is the *Transfer efficiency* so in the rest of this section we focus on studying it using the case with 512 processes (*Transfer efficiency* below 80%). The remaining metrics with low values are left out of the study due to space constraints.

	64	128	256	512	1024
Global efficiency	85.47	80.97	71.94	62.21	37.38
├ Parallel efficiency	85.47	81.69	72.81	61.95	43.33
│ ├ Load balance	91.97	92.23	91.87	89.52	86.57
│ └ Communication eff.	92.93	88.57	79.25	69.20	50.05
│ ├ Serialization eff.	95.50	93.80	91.01	89.51	83.60
│ └ Transfer eff.	97.31	94.43	87.08	77.31	59.06
└ Computation scalability	100.00	99.12	98.81	100.43	86.28
├ IPC scalability	100.00	100.63	103.26	105.92	114.83
├ Instruction scalability	100.00	98.48	95.74	94.89	75.49
└ Frequency scalability	100.00	100.03	99.94	99.92	99.52

100% 95% 90% 85% <80%

Fig. 7: Multiple-node efficiency metrics of OpenFOAM.

We isolated a section of the iteration corresponding to 36% of the total iteration time where the *Transfer efficiency* drops below 50% and the density of MPI calls is high: 63% of the MPI time of the iteration is concentrated in this section.

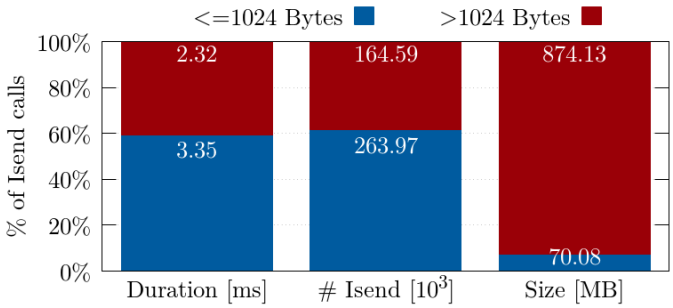

Fig. 8: Duration, number of calls and bytes exchanged with small `Isend`.

Figure 8 shows a study of `Isend`, the MPI call that moves most of the data among processes. We report the time spent, number of calls and the number of byte transmitted for all messages and for messages smaller or equal than 1024 bytes. We observe that to exchange 7% of the bytes the application spends 59% of the time performing all `Isend` calls. Moreover we note that 61% of the `Isend` calls are moving 1024 bytes or less. We conclude that the low *Transfer efficiency* is due to the combination of two factors. On one hand, the high number of MPI calls sending small messages. On the other hand, the MPI overhead as well as network latency that are paid for each call.

VII. CP2K

CP2K is a framework for atomistic simulations written in Fortran and is part of the Unified European Applications Benchmark Suite[5].

This application uses GNU intrinsics which are incompatible with the Arm HPC compiler. Hence, we used the GNU compiler suite for our studies. CP2K depends on BLAS and LAPACK and it can take advantage of FFTW to improve the performance of FFT operations. We used the Arm Performance Libraries to cover these software dependencies. The rest of the dependencies of CP2K are supplied with the package itself and we used the minimum subset of them to support our input set.

The input set used in our study uses CP2K's Molecular Dynamics (MD) module. Specifically, the Quantum Mechanics / Molecular Mechanics (QMMM). The simulation consists of three distinct phases, FIST, QS/DFTB and QMMM. We focused our analysis on the QMMM phase which is the most time consuming part of the execution. QMMM performs the simulation of the system iterating through multiple steps. Each step invokes an iterative convergence method called Self-Consistent Field. Our case study sets a cut-off of 37 iterations. Since all of them present a similar computational footprint, we limited our study to four of them.

[5] http://www.prace-ri.eu/IMG/pdf/Selection_of_a_Unified_European_Application_Benchmark_Suite.pdf

A. *Single-node performance analysis*

Figure 9 shows the efficiency metrics of CP2K when running on one node of Dibona. We observe a dramatic drop in the *Computation scalability* due to the *Instruction scalability*. Meaning that the amount of instructions executed by each MPI process does not decrease proportionally when increasing the number of MPI processes.

	2	4	8	16	32	64
Global efficiency	99.67	64.58	37.53	20.93	10.85	5.48
├ Parallel efficiency	99.67	98.60	98.97	99.02	98.87	98.14
│ ├ Load balance	99.94	99.07	99.83	99.72	99.70	98.95
│ └ Communication eff.	99.74	99.53	99.14	99.30	99.16	99.18
│ ├ Serialization eff.	99.88	99.67	99.31	99.47	99.38	99.45
│ └ Transfer eff.	99.86	99.86	99.83	99.82	99.78	99.73
└ Computation scalability	100.00	65.49	37.92	21.14	10.97	5.58
├ IPC scalability	100.00	95.55	92.23	89.26	88.66	87.72
├ Instruction scalability	100.00	68.76	41.13	23.67	12.38	6.38
└ Frequency scalability	100.00	99.68	99.97	100.08	99.99	99.83

100% 95% 90% 85% <80%

Fig. 9: Single-node efficiency metrics of CP2K

To identify the cause of the bad *Instruction scalability*, we used the clustering tool introduced in Section IV-B. This allowed us to identify computational regions with similar behavior (called for brevity *clusters*) and study them independently. The identified clusters cover about 90% of the execution time regardless of the number of processes.

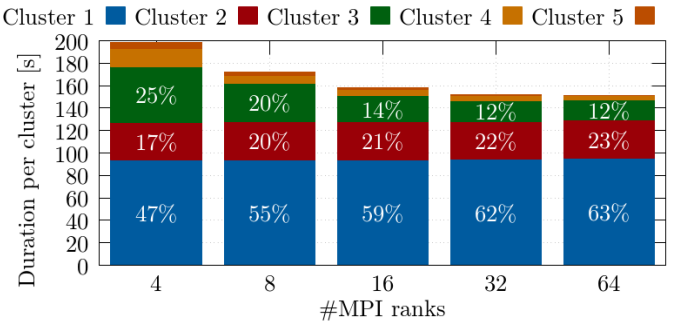

Fig. 10: Duration per cluster in CP2K

Figure 10 shows the total duration of each cluster varying the number of MPI processes. The y-axis represents time in seconds and each cluster is color-coded. The duration of clusters one and three do not scale at all disregarding the number of MPI processes.

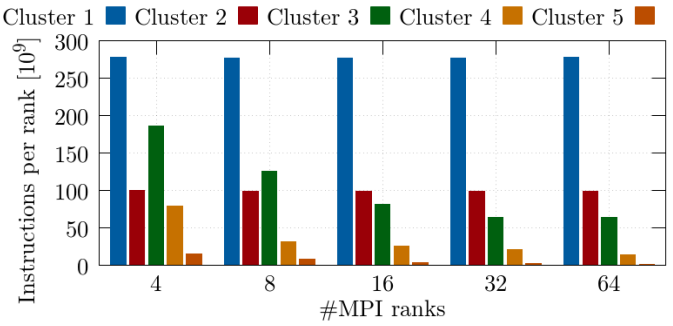

Fig. 11: Instructions per cluster in CP2K

Figure 11 shows the total number of instructions per rank executed in each cluster. The x-axis represents the number of processes and the y-axis represents the number of executed instructions per rank. Each cluster is color-coded. We confirm that cluster one and two are responsible for the poor *Instruction scalability*: the number of instructions executed by each MPI process remains constant. The duration and the number of instructions of clusters three and four scale proportionally only up to 16 MPI processes. When using more than 16 MPI processes scalability becomes bad also for these clusters. The next natural step would be to map the clusters in code regions to address the parallelization leading to the bad *Instruction scalability*. We leave this as future work or a hint for the CP2K developers.

B. *Multi-node performance analysis*

	64	128	256	512	1024
Global efficiency	98.27	49.61	24.75	12.14	4.94
├ Parallel efficiency	98.27	98.00	97.30	95.67	91.93
│ ├ Load balance	98.99	99.27	98.90	98.10	98.75
│ └ Communication eff.	99.27	98.72	98.38	97.52	93.09
│ ├ Serialization eff.	99.54	99.49	99.48	99.35	98.47
│ └ Transfer eff.	99.73	99.22	98.89	98.16	94.53
└ Computation scalability	100.00	50.62	25.44	12.69	5.37
├ IPC scalability	100.00	98.72	98.66	98.67	100.73
├ Instruction scalability	100.00	51.22	25.76	12.85	5.34
└ Frequency scalability	100.00	100.11	100.09	100.08	99.97

100% 95% 90% 85% <80%

Fig. 12: Multiple-node efficiency metrics of CP2K.

In Table 12, we show the multi-node efficiency figures for CP2K. As with the single-node study, we observe that the main limiting factor is the *Computation scalability*. However, we also notice a steady decrease in *Transfer efficiency* when increasing the number of processes. This metric is likely to become the main limiting factor when scaling beyond 16 nodes.

MPI processes	**256**	**512**	**1024**
% of MPI time in `Alltoallv` calls	15.00%	21.51%	43.04%
% of MPI time in `Allreduce` calls	59.47%	58.17%	32.09%
% of MPI time in `Alltoall` calls	11.28%	7.21%	10.15%
Avg. message size of `Alltoallv` calls	6.35 KiB	3.47 KiB	1.72 KiB
% of MPI time in `Alltoallv` transfer	0.38%	0.85%	3.37%

TABLE II: CP2K MPI statistics when scaling ≥ 4 nodes.

We analyzed the most called MPI primitives and realized that they were collective operations. In Table II we show the percentage of MPI time of the three most relevant MPI calls with 256, 512 and 1024 processes. We observe that the `Allreduce` is the MPI primitive that represents the most time with 256 processes but `Alltoallv` takes its spot when scaling up to 1024 processes.

The third row of Table II displays the average message size of `Alltoallv` calls. We observe that the average message size decreases when increasing the number of processes. As with OpenFOAM in Section VI, overheads of the MPI implementation when exchanging small messages can be the cause of the decrease in *Transfer efficiency*. Moreover, collective calls with a high number of processes lead to network congestion.

Lastly, the fourth row in Table II shows the time of the actual exchange of data in the `Alltoallv` calls with respect to the total time spent in MPI calls. We measured this time as the duration of the MPI call of the process that reached last the communication. Our measurements show that with 256 and 512 processes, the transfer time of all `Alltoallv` calls represents under 1% of the total execution time. In contrast, with 1024 processes, the transfer time reaches 3.37% of the total execution time.

We conclude that the *Transfer efficiency* of CP2K keeps decreasing when scaling beyond 16 nodes because of the high number of collective calls with small messages. A possible solution would be to decrease the number of collective operations by aggregating them into a single MPI call.

VIII. Grid

Grid is an open-source data-parallel C++ mathematical object library that implements data structures aware of SIMD architectures [13]. We ran Grid in parallel using MPI. The program used in our study is a compressed Lanczos hot start algorithm provided by the Grid repository that makes extensive use of the Grid library.

A. Single-node performance analysis

Figure 13 shows the POP metrics of Grid within a single compute node. We recognize that the main limiting factor of the *Global scalability* is the *IPC scalability*, which decreases when the number of processes increases up to a full node.

	8	16	32	64
Global efficiency	99.32	94.38	78.87	77.58
├ Parallel efficiency	99.32	99.09	93.53	93.90
│ ├ Load balance	99.61	99.76	99.40	99.28
│ └ Communication eff.	99.71	99.32	94.09	94.58
│ ├ Serialization eff.	99.73	99.36	94.15	94.71
│ └ Transfer eff.	99.98	99.96	99.93	99.86
└ Computation scalability	100.00	95.25	84.33	82.62
├ IPC scalability	100.00	95.22	84.43	82.63
├ Instruction scalability	100.00	100.02	100.03	99.99
└ Frequency scalability	100.00	100.01	99.85	100.00

100% 95% 90% 85% <80%

Fig. 13: Single-node efficiency metrics of Grid.

Using the *Clustering* tool introduced in Section IV-B, we distinguished six regions of code with a similar IPC (also called *clusters*). The amount of compute time spent in these clusters is more than 80% regardless of the number of processes, so they are representative of the actual workload performed by the application. For each cluster, we studied the values of IPC when scaling the number of processes. In Figure 14 we plot on a plane x="IPC", y="Number of useful instructions executed per process" how the centroid of each cluster moves when changing the number of MPI processes. Therefore, the ideal situation would be to have vertical lines, meaning, the number of instructions executed by process is reduced proportionally to the number of processes and the IPC is constant. Our plot shows that three clusters are responsible for the IPC drop (clusters 4, 5, and 6), i.e. clusters "moving" to the left. Moreover cluster 4 and 5 show a more abrupt IPC drop when scaling from 16 to 32 MPI processes.

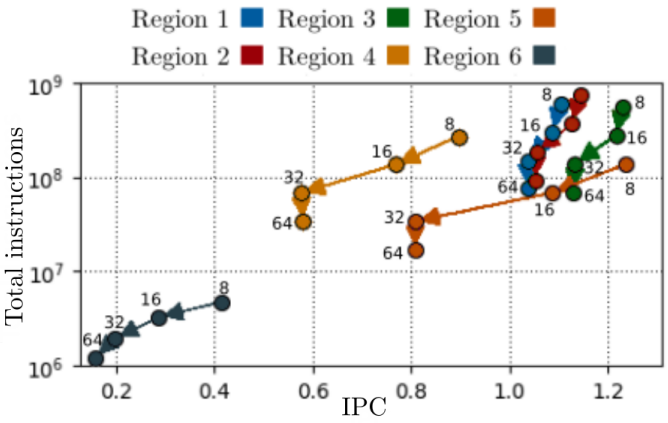

Fig. 14: Evolution of IPC per cluster in Grid

The natural next step would be to identify the regions of code represented by each cluster and try to improve their IPC in collaboration with Grid developers. In the interest of space we left this out of the paper.

B. Multi-node performance analysis

Figure 15 shows the POP metrics of Grid when running on multiple nodes of the Dibona cluster. We can see that the two factors affecting the performance of Grid are *Instruction scalability* and *Transfer efficiency*.

	64	128	256	512	1024
Global efficiency	93.90	86.83	81.69	73.76	52.67
├ Parallel efficiency	93.90	88.62	83.62	78.07	68.98
│ ├ Load balance	99.28	98.54	98.27	97.89	93.20
│ └ Communication eff.	94.58	89.93	85.09	79.75	74.01
│ ├ Serialization eff.	94.71	94.34	93.04	91.58	90.53
│ └ Transfer eff.	99.86	95.32	91.46	87.08	81.75
└ Computation scalability	100.00	97.98	97.69	94.49	76.37
├ IPC scalability	100.00	98.10	98.93	103.53	134.15
├ Instruction scalability	100.00	99.84	98.70	91.04	56.67
└ Frequency scalability	100.00	100.04	100.05	100.25	100.44

100% 95% 90% 85% <80%

Fig. 15: Multiple-node efficiency metrics of Grid.

To study the *Instruction scalability* once more we used the *Clustering* tool and identified the same six representative regions that we studied when running on a single node.

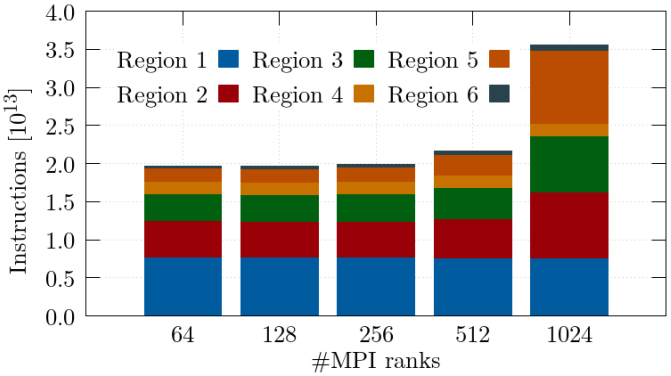

Fig. 16: Number of instructions by cluster in an iteration.

In Figure 16, we report the total instructions (y-axis) executed by the application for each number of MPI processes (x-axis) and each cluster is color coded. It is clearly visible that clusters 2, 3, and 5 are responsible for the increment of instructions at the highest number of MPI ranks. At this point a further study would allow us to relate the clusters responsible to the source code to try to solve the parallelization issue.

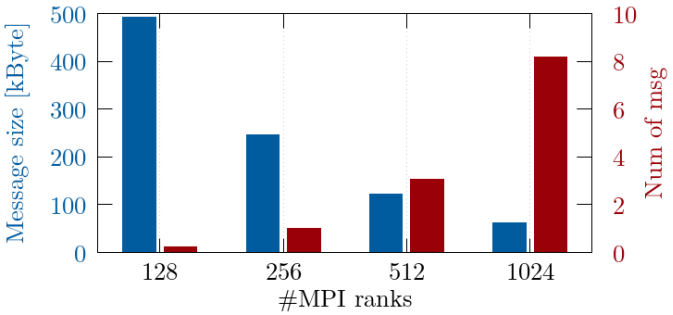

Fig. 17: Message size vs num. of messages in Grid.

The other efficiency metric with an alarming value is the *Transfer efficiency*. To find the cause of its low value, we analyze the communication pattern of the application. We observe that communication is mainly done with MPI point-to-point calls (i.e. `ISend` and `Wait`).

In Figure 17 we show the number of `ISend` calls and their average message size. We can observe that the size of the messages sent decreases and the number of point-to-point messages increases when using more MPI processes. Due to the MPI overhead, and network latency this trend translates into a more inefficient communication. The suggestion for Grid developers would be to try to reduce the number of messages sent and increase its size by fusing messages when possible.

IX. Conclusions

In this paper we analyzed the performance of four parallel HPC applications using a well established method leveraging different efficiency metrics for guiding the analysis. We performed our study on a state-of-the-art Arm-based cluster based on Marvell Cavium ThunderX2, the same CPU technology powering the Astra supercomputer recently ranked in the Top500 list. The aim of this section is to provide lessons learned in our study process and co-design insights, i.e., guidelines for both system designers and application developers.

We show the power of having objective metrics such as the ones defined by the POP Center of Excellence that we can easily apply to different workflows. This allows us not only to spot parallel inefficiencies due to poor parallelization strategies but also to identify problems in the hardware and software configuration of an HPC cluster. Also, since we applied the method leveraging POP metrics to data gathered on a cluster powered by Arm-based CPUs, we proved that such a method delivers effective insights independently on the architecture.

When deploying an HPC cluster, the tuning of the system software is of paramount importance. We noticed, e.g., with Pennant, that OS noise and MPI implementation can heavily affects the performance of parallel applications. A complete set of hardware counters accessible via a standard interface, e.g., PAPI, is extremely important to identify the root cause of drops in the POP metrics. For example, the lack of the L3 cache miss counter in the Dibona cluster did not allow us to confirm the suspect of saturation of the memory bandwidth with OpenFOAM.

Concerning HPC application development and parallelization strategies, we also highlighted that scaling to multiple processes is not beneficial if there are regions of code that do not scale. This is kind of obvious, but in CP2K we proved that this behavior heavily affects the performance of the application. Also, the study performed on OpenFOAM demonstrate that collective operations among several processes can become a serious scalability limitation, with a higher impact when the amount of data exchanged among processes is small.

Acknowledgment

This work is partially supported by the Spanish Government (SEV-2015-0493), by the Spanish Ministry of Science and Technology (TIN2015-65316-P), by the Generalitat de Catalunya (2017-SGR-1414), and by the European Mont-Blanc 3 project (GA n. 671697) and POP CoE (GA n. 824080).

References

[1] F. Banchelli and F. Mantovani, "Filling the gap between education and industry: evidence-based methods for introducing undergraduate students to hpc," in *2018 IEEE/ACM Workshop on Education for High-Performance Computing (EduHPC)*. IEEE, 2018, pp. 41–50.

[2] C. Niethammer *et al.*, *Tools for High Performance Computing 2017: Proceedings of the 11th International Workshop on Parallel Tools for High Performance Computing*. Springer, 2017.

[3] M. Burtscher *et al.*, "Perfexpert: An easy-to-use performance diagnosis tool for hpc applications," in *SC'10: Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2010, pp. 1–11.

[4] L. Stanisic *et al.*, "Performance analysis of hpc applications on low-power embedded platforms," in *2013 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2013, pp. 475–480.

[5] M. Garcia-Gasulla *et al.*, "Runtime mechanisms to survive new HPC architectures: A use case in human respiratory simulations," *The International Journal of High Performance Computing Applications*, 2019.

[6] E. Calore *et al.*, "Advanced performance analysis of HPC workloads on Cavium ThunderX," in *2018 International Conference on High Performance Computing & Simulation (HPCS)*, 2018, pp. 375–382.

[7] S. McIntosh-Smith, J. Price, T. Deakin, and A. Poenaru, "A performance analysis of the first generation of hpc-optimized arm processors," *Concurrency and Computation: Practice and Experience*, p. e5110, 2018.

[8] F. Banchelli *et al.*, "MB3 D6.9 – Performance analysis of applications and mini-applications and benchmarking on the project test platforms," Tech. Rep., 2019. [Online]. Available: http://bit.ly/mb3-dibona-apps

[9] M. Wagner, S. Mohr, J. Giménez, and J. Labarta, "A structured approach to performance analysis," in *International Workshop on Parallel Tools for High Performance Computing*. Springer, 2017, pp. 1–15.

[10] H. Servat *et al.*, "Framework for a productive performance optimization," *Parallel Computing*, vol. 39, no. 8, pp. 336–353, 2013.

[11] V. Pillet, J. Labarta, T. Cortes, and S. Girona, "Paraver: A tool to visualize and analyze parallel code," in *Proceedings of WoTUG-18: transputer and occam developments*, vol. 44, no. 1, 1995, pp. 17–31.

[12] C. R. Ferenbaugh, "Pennant: an unstructured mesh mini-app for advanced architecture research," *Concurrency and Computation: Practice and Experience*, vol. 27, no. 17, pp. 4555–4572, 2015.

[13] P. Boyle, A. Yamaguchi, G. Cossu, and A. Portelli, "Grid: A next generation data parallel c++ qcd library," *arXiv:1512.03487*, 2015.