The final publication is available at ACM via http://dx.doi.org/10.1145/3394277.3401847

Benchmarking of state-of-the-art HPC clusters with a production CFD code

Fabio Banchelli
fabio.banchelli@bsc.es
Barcelona Supercomputing Center
Barcelona, Spain

Marta Garcia-Gasulla
marta.garcia@bsc.es
Barcelona Supercomputing Center
Barcelona, Spain

Guillaume Houzeaux
guillaume.houxeaux@bsc.es
Barcelona Supercomputing Center
Barcelona, Spain

Filippo Mantovani
filippo.mantovani@bsc.es
Barcelona Supercomputing Center
Barcelona, Spain

ABSTRACT

Computing technologies populating high-performance computing (HPC) clusters are getting more and more diverse, offering a wide range of architectural features. As a consequence, efficient programming of such platforms becomes a complex task. In this paper we provide a micro-benchmarking of three HPC clusters based on different CPU architectures, predominant in the Top500 ranking: x86, Armv8 and IBM Power9. On these platforms we study a production fluid-dynamics application leveraging different compiler technologies and micro-architectural features. We finally provide a scalability study on state-of-the-art HPC clusters. The two most relevant conclusions of our study are: *i)* Compiler development is critical for squeezing performance out of most recent technologies; *ii)* Micro-architectural features such as Single Instruction Multiple Data (SIMD) units and Simultaneous Multi-Threading (SMT) can impact the overall performance. However, a closer look shows that while SIMD is improving the performance of compute bound regions, SMT does not show a clear benefit on HPC workloads.

CCS CONCEPTS

• **Applied computing** → *Physics*; • **Computing methodologies** → *Parallel computing methodologies*; • **General and reference** → **Performance**; • **Computer systems organization** → *Multicore architectures.*

KEYWORDS

Cluster computing, High Performance Computing, Computational Fluid Dynamics, Simultaneous Multi-Processing, Vectorization, SIMD, Compilers, Parallel Applications, x86, Arm, ThunderX2, IBM Power9

ACM Reference Format:
Fabio Banchelli, Marta Garcia-Gasulla, Guillaume Houzeaux, and Filippo Mantovani. 2019. Benchmarking of state-of-the-art HPC clusters with a production CFD code. In *Proceedings of* . ACM, New York, NY, USA, 11 pages.

, ,

1 INTRODUCTION AND RELATED WORK

High-Performance Computing (HPC) data centers are full of diverse CPU architectures. New architectures powering modern HPC compute nodes are complex: they often hide underlying micro-architectures, complex cache hierarchies, vector units with relatively large vector size, etc. To take advantage of these features and reduce the complexity exposed to the customers, technology providers often make available to system integrators and final users system software tools (compilers, libraries, runtimes, etc.). While this is, in general, a good strategy for the user who can relatively easily migrate to new systems, it is difficult to estimate the impact of such a combination of hardware features and software tricks. For this reason, the first part of our paper is dedicated to the characterization of three HPC state-of-the-art clusters.

The three HPC cluster considered for this study are: *MareNostrum4*[1], is a Tier-0 supercomputer composed of 3456 homogeneous compute nodes powered by Intel Skylake CPUs. *Power9*[2], a small clone of Summit [27], the top-ranked system of the Top500. The Power9 cluster is composed of 50 compute nodes powered by latest IBM Power9 CPUs (plus 4 Volta GPUs per node that are not used for our studies in this paper); *Dibona*, a production-ready cluster [1] composed of 48 compute nodes powered by Marvell ThunderX2 CPUs (using the Armv8 architecture). Bull/ATOS developed it during the third phase of the Mont-Blanc project [22].

In the literature we can find several examples of tuning benchmarks and proxy applications on such new architectures [2, 3, 15, 17, 24]. Also, the literature offers numerous analysis of large HPC systems, but often focusing on specific components, e.g., CPUs [9] or memory usage [16] or particular workloads, like Kahle et al. in [8]. However, scientists typically use complex domain-specific applications for their studies on large production HPC clusters. In our research, we consider a real use case consisting of the computational fluid dynamics (CFD) simulation of a full airplane using Alya [28].

Alya is a multi-physics code for solving mainly partial differential equations with the finite element method on unstructured meshes. It is written in Fortran and is parallelized with MPI and OpenMP [7]. There exist mainly two numerical methods to solve partial differential equations on unstructured meshes, namely the

[1] https://www.bsc.es/marenostrum/marenostrum
[2] https://www.bsc.es/user-support/power.php

finite volume and the finite element method [13]. In both cases, CFD codes involve two phases: the assembly of vectors and possibly matrices, and the iterative solvers to solve the resulting algebraic systems, if required. The assembly phase consists of a loop over some geometric entities of the computational mesh, namely faces, elements, or edges. In the case of explicit schemes, no iterative solver is required so that the computational performance of the code relies exclusively on the assembly [14]. In the present context, Alya is element-based and the solution scheme used is semi-implicit fractional step method [12]. The iterative solver considered is the conjugate gradient [25].

The main contributions of this paper are: *i)* a performance comparison of three clusters based on x86, Armv8, and IBM Power9 architectures. *ii)* an in-depth evaluation of a real scientific application representative of HPC workloads looking at compilers and multi-threading; *iii)* a scalability study of a production CFD use case running on state-of-the-art HPC architectures.

The remaining part of the document is structured as follows: Section 2 introduces the scientific code used in our study. In Section 3 we provide the low-level characterization of the HPC clusters used in our evaluation. Section 4 focuses on the evaluation of vendor-specific and open-source software tools (mostly compilers), enabling underlying hardware features (mostly vector units). Section 5 analyzes the benefits of using the Simultaneous Multi-Threading (SMT) on each of the considered clusters. We complete our study presenting in Section 6 the performance at scale of Alya on the three HPC clusters considered in this paper. We close the paper with our comments and consideration in Section 7.

2 APPLICATION CHARACTERIZATION

In this section, we describe the computational fluid dynamic application used throughout our study. We start describing the use case and the physical problem, we then discuss the numerical strategy employed, and finally, we identify and characterize different phases within the execution.

2.1 Test case: full airplane simulation

The selected application on which we carried out the experiments of this paper, is a full airplane configuration, at a Mach number of 0.172, with a Reynolds number based on the mean aerodynamic chord of 1.93 M and an angle of attack of 18.58 degrees [31, 32]. The mesh is a linear hybrid mesh consisting of 31.5 million elements (with tetrahedra, prisms, and pyramids). It has approximately three elements in the inner region of the boundary layer and about 10 elements in the out layer region. In this particular case, the percentage of boundary elements over volume elements is 5%.

Physical problem. The governing equations to solve the proposed physical problem are the filtered incompressible Navier-Stokes equations. They solve for the fluid velocity $\boldsymbol{u}$ and pressure p of the fluid during a given time interval such that

$$\rho\frac{\partial \boldsymbol{u}}{\partial t} + \rho\left[2\mathbf{u}\cdot\boldsymbol{\varepsilon}(\mathbf{u}) + (\nabla\cdot\mathbf{u})\,\mathbf{u} - \frac{1}{2}\nabla|\mathbf{u}|^2\right] - \nabla\cdot[2\mu\boldsymbol{\varepsilon}(\boldsymbol{u})] + \nabla p + \nabla\cdot\boldsymbol{\tau} = \mathbf{0}, \quad (1)$$

$$\nabla\cdot\boldsymbol{u} = 0, \quad (2)$$

together with initial and boundary conditions. The velocity strain rate is defined as $\boldsymbol{\varepsilon}(\boldsymbol{u}) := \frac{1}{2}(\nabla\boldsymbol{u} + \nabla\boldsymbol{u}^t)$, and $\boldsymbol{\tau}$ is the subgrid scale stress tensor [21]. At the continuous level, the convective term (second term in brackets in Equation 1) can be written in many equivalent ways. In this work, we consider the EMAC formulation [4, 12] which, at the discrete level, conserves energy as well as linear and angular momentum.

To model the subgrid scale stress tensor, Vreman LES model is used [29]. Wall modeling is based on a law of the wall, extensively described by Owen et al. in [19].

Numerical strategy. The code considered in this work, Alya, is an element-based finite element solver. Its space discretization is carried out using the Galerkin Finite element method [6, 13, 23]. Stability of the momentum equation is provided by the EMAC formulation, avoiding extra stabilization terms in these equations. We use a fractional step method together with a Runge-Kutta scheme of third order as a time discretization scheme. Momentum is advanced explicitly, while the projection step of the algorithm, which consists of solving a Poisson equation for the pressure, provides the pressure stabilization [5]. This associated algebraic system is solved with the conjugate gradient method [25]. Lehmkuhl et al. [12] provide details on the numerical and time integration schemes.

Solution procedure. At each time-step, three Runge-Kutta steps are performed. Each of these Runge-Kutta steps, involves two main assembly loops:

- *Element assembly*: a loop over the elements to compute a vector, representing the momentum equation residual;
- *Boundary assembly*: a loop over the boundaries which consist of the element external faces, in order to assemble the boundary conditions (law of the wall) into the momentum equation residual.

Finally, at the end of the Runge-Kutta steps, the conjugate gradient method is used to solve the pressure Poisson equation [25] with an in-house implementation. In this paper, this phase is referred to as *iterative solver*.

2.2 Computational Phases

Alya is fully parallelized with MPI. The mesh partitioning is achieved using METIS [10], minimizing the number of neighbors.

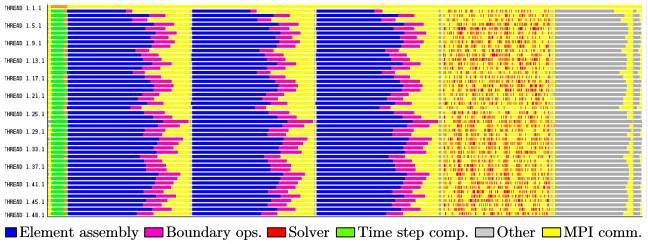

Figure 1: Timeline of one time step of the simulation

To identify the different computational phases of Alya we used Extrae [26] to obtain a trace with 48 MPI processes and visualize it using Paraver [20]. In Figure 1 we can see a timeline of one time step, each horizontal line represents one MPI process, the x axis

shows the time. The color code represents the activity of each MPI process at a given point in time. The different phases identified are: the three Runge-Kutta steps, each one including the *Element assembly* (blue) and the *Boundaries operations* (pink), the only MPI communication in these steps is a global synchronization at the end of each step. We can observe that one of the main issues of both phases is the load balance between MPI processes. Moreover, the load balance depends on the partition and the geometry. Also, depending on the partition, some MPI processes may not have any boundary element to compute and this issue worsens when increasing the number of partitions (MPI processes).

The algebraic *Solver* (red), on the other hand, includes a high number of communications. Each iteration of the solver needs to perform several MPI communications, including point-to-point and global reduction operations, as explained in [28].

In the example shown, the element assembly phase accounts for 40% of the time of the time step, the boundary operations 10%, and the solver 12%. The reader should note that depending on the macroscopic problem simulated by Alya, the underlying microscopic parameters of the simulations (e.g., number of elements, boundary loops or iterations required by the solver) can change significantly. This means that the three phases can have different relative durations.

3 CLUSTER CHARACTERIZATION

In this section, we describe the three state of the art HPC platforms used in our study: Dibona, MareNostrum4, and Power9. In Table 1 we summarize their hardware and software configurations. Turboboost, as well as dynamic frequency scaling, have been consistently disabled for all tests of our study.

Table 1: Hardware configuration of the HPC platforms

	Dibona	MareNostrum4	Power9
CPU name	Marvell ThunderX2	Skylake Platinum	IBM Power9 8335-GTH
Core architecture	Armv8	Intel x86	Power ISA v3.0B
Frequency [GHz]	2.0	2.1	3.0
Sockets/node	2	2	2
Core/socket	32	24	20
L1 cache	private 32 KiB	private 32 KiB	private 32 KiB
L2 cache	private 256 KiB	private 1 MiB	shared 512 KiB
L3 cache	shared 32 MiB	private 33 MiB	shared 10 MiB
Hw threads/core	up to 4	up to 2	up to 4
Memory/node [GB]	256	96	512
Memory tech.	DDR4-2666	DDR4-2666	DDR4-2666
Memory channels	8	6	8
Num. of nodes	40	3456	50
Interconnection	Infiniband EDR	Intel OmniPath	Infiniband EDR
OS	RHEL 7.5	Suse 12 SP2	RHEL 7.5
MPI	OpenMPI 3.1.2	OpenMPI 3.1.1	OpenMPI 3.1.1

The Dibona cluster stems from the European project Mont-Blanc 3 [1]. The cluster is integrated using the ATOS/Bull Sequana infrastructure and comprises 40 compute nodes. The CPUs of the Dibona cluster are identical to Astra [11], the first Arm-based supercomputer ranked in the Top500 list (156th in June 2019). All data related to Dibona presented in the rest of the paper are color-coded in red and labeled as *mb3*.

The MareNostrum4 Supercomputer is a Tier-0 supercomputer ranked 29th in the Top500 (June 2019) with a total of 3456 x86 compute nodes. All data related to MareNostrum4 presented in the rest of the paper are color-coded in blue and labeled as *mn4*.

The Power9 cluster has a total of 50 compute nodes based on the IBM Power9 architecture. The architecture of the Power9 cluster used in this paper is identical to Summit [27], the supercomputer ranked first in the Top500 list (June 2019). Also, Power9 is ranked 5th in the Green500 list (June 2019). All data related to Power9 presented in the rest of the paper are color-coded in green and labeled as *p9*.

3.1 Roofline model

The roofline model [30] expresses the floating-point performance as a function of the arithmetic intensity, modeled as the number of floating point operations executed on each byte transferred to/from memory. This function can be represented as a line on a two-dimensional plane where the x-axis is the arithmetic intensity, and the y-axis is the floating point performance.

$$F_s(I) = \begin{cases} F_p & \text{if } I > I_r \\ B_p \cdot I & \text{if } I \leq I_r \end{cases} \tag{3}$$

The Equation 3 shows the theoretical formulation of the roofline model, giving the sustained floating point performance F_s as a function of the arithmetic intensity I. F_p and B_p are respectively the peak floating point performance (expressed in Flop/s) and the peak memory bandwidth to/from the memory (expressed in Byte/s), while $I_r = F_p/B_p$. We notice that $F_s(I)$ as defined in Equation 3 only depends on the peak performance of the floating point unit and the memory.

We base our studies on the work by Ofenbeck et al. [18], where the authors describe a methodology to construct the roofline model empirically using micro-benchmarks. Using a similar approach, we derive the roofline model of the three HPC clusters under study. We measure the memory bandwidth with the STREAM benchmark and the floating-point performance with a custom micro-kernel so to be able to compute $F_s(I)$.

Memory bandwidth. To measure the memory bandwidth of one node, we use the STREAM benchmark (v5.10.1), leveraging an OpenMP parallelization. The problem size is fixed on each system to be at least four times the size of the sum of all the last-level caches. We run STREAM increasing the number of OpenMP threads and set the environment variable `OMP_BIND_PROC=true` to ensure that all threads are pinned to a core. When not using the maximum number of threads per node, threads are evenly distributed across both sockets. Therefore, we minimize the number of threads accessing the same shared resources (i.e., L2 and L3 caches). Figure 2 shows the results of our studies. We report only the Triad kernel as a representative of a typical HPC workload. The x-axis represents the number of OpenMP threads, and the y-axis indicates the maximum achieved bandwidth of 200 executions. The horizontal lines indicate the theoretical peak bandwidth for each system.

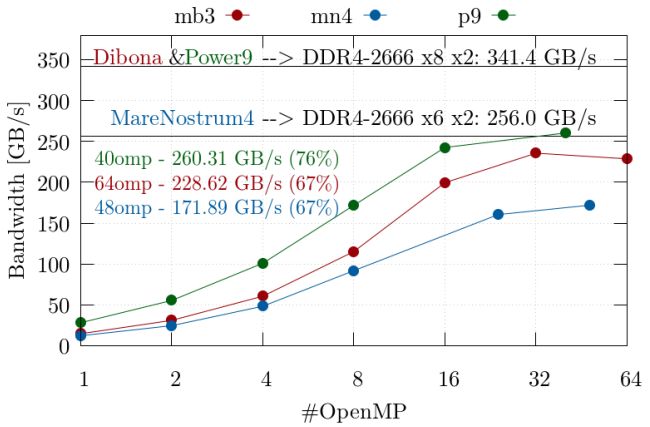

Figure 2: STREAM Triad on one node of Dibona, MareNostrum4 and Power9

Floating point peak performance. To measure the sustained floating-point performance, we use a micro-kernel that stresses the floating-point and SIMD units executing a loop of fused-multiply-and-add instructions with no data dependency between them. Table 2 shows the assembly instructions used in the micro-kernel. We report only double precision performance because Alya does not use single precision data types.

Table 2: Instructions in the FPU_μKernel

Machine	Instruction	Type
Dibona	`FMADD Dx,Dx,Dx,Dx`	FPU DP
MareNostrum4	`VFMADD132SD mmx,mmx,mmx`	FPU DP
Power9	`FMADD x,x,x,x`	FPU DP
Dibona	`FMLA Vx.2D,Vx.2D,Vx.2D`	SIMD DP
MareNostrum4	`VFMADD132PD mmz,mmz,mmz`	SIMD DP
Power9	`XVMADDADP x, x, x`	SIMD DP

Figure 3 shows the results of the FPU_μKernel on one core of each machine. It is important to note that the MareNostrum4 nodes have a SIMD unit of 512-bits while the SIMD registers of Dibona and Power9 are 128-bits wide.

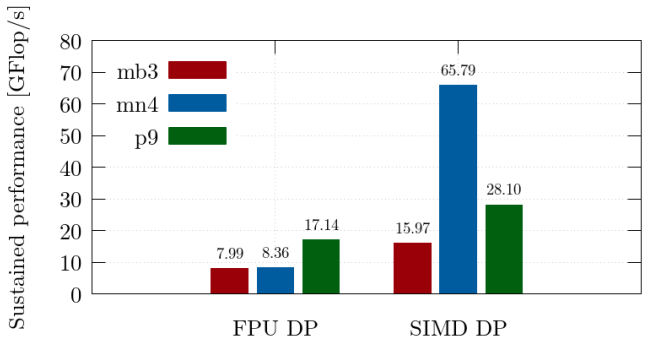

Figure 3: FPU_μKernel on one core of Dibona, MareNostrum4 and Power9

Roofline model. From Figure 2 and 3, we can extract the peak memory bandwidth B_p and the peak floating point performance F_p so we can construct the roofline curves for one node of each of our clusters. Solid lines in Figure 4 plot $F_s(I)$ expressed in Equation 3 for a single compute node of each of the clusters. The x-axis represents the arithmetic intensity expressed in byte/Flop and the y-axis represents the theoretically achievable peak performance Flop/s.

The section of the curve with a non-zero slope identifies a range of values of arithmetic intensity where the memory bandwidth limits the computational throughput. The bandwidth of the system gives the slope of the curve. On the other hand, the flattened part of the curve represents the region, where the limiting factor is only the floating point throughput. For drawing the computational limit of each cluster, we consider the sustained performance delivered by the SIMD unit when working with double precision data types (right-most data set of Figure 3). For the sake of simplicity, we only considered the bandwidth to the main memory (ignoring the bandwidth delivered by caches) and the computational power provided by SIMD instructions running at the nominal frequency reported in Table 1.

4 COMPILER COMPARISON

As mentioned in Section 1, software tools (e.g., compilers, libraries, or runtimes) are necessary to hide the underlying complexity of modern CPUs from end-users. These tools sometimes are provided by the CPU vendors. Therefore, one would expect that they will be able to exploit the CPU performance at its maximum. But they can also be provided by third parties or the open-source community.

In this section, we study the performance of Alya across our three HPC clusters using different compilers, both from the open-source community and from CPU vendors. For this study, we employ four metrics to evaluate the performance delivered by the various compilers in each phase: arithmetic intensity, computational performance (or directly GFlop/s), Instructions per Cycle (IPC) and autovectorization.

4.1 Methodology

Table 3 shows the list of the compilers available on each cluster as well as the optimization flags used for each case of our study. All compilations were performed using the `-O3` flag. All runs of this section have been executed running the MPI-only version of Alya filling all cores of one compute node of each cluster.

We used Extrae [26] to collect data from the PAPI library during the execution of Alya and generate a trace. We then used Paraver [20] to visualize the trace and extract the metrics described above. We measured that the overhead introduced by Extrae is always below 5% compared to the execution time without tracing. The Extrae instrumentation tool leverages events triggered by the application, e.g., the MPI calls, to gather information about the running code, e.g., it calls the underlying PAPI library to collect data from hardware counters from each of the MPI processes. Each interval of time in which Extrae gathers information is called *burst*. Each phase p of Alya introduced in Section 2.2, is composed of B bursts while P is the number of MPI processes.

For each burst of each MPI process, we collect f, the number of floating point operations executed in that burst, m, the number of

Table 3: Compiler flags and PAPI counters used on each HPC cluster

Machine	Compiler	Flags	f [Flop]	m [Bytes]	Vector instructions
Dibona	GNU 8.1.0	`-mcpu=thunderx2t99 -ffp-contract=fast -ffast-math`	`PAPI_FP_INS` + 2 * `PAPI_VEC_INS`	64 * `PAPI_L2_DCM`	`PAPI_VEC_INS`
	Arm HPC Compiler 19.0	`-mcpu=thunderx2t99 -ffp-contract=fast -ffast-math`			
MareNostrum4	GNU 8.1.0	`-march=skylake-avx512 -ffp-contract=fast -ffast-math`	`PAPI_DP_OPS`	64 * `PAPI_L3_TCM`	`PAPI_VEC_DP`
	Intel Compiler 2017.4	`-xCORE-AVX512 -mtune=skylake -heap-arrays -ipo`			
Power9	GNU 8.1.0	`-mtune=power9 -mcpu=power9 -maltivec`	`PAPI_DP_OPS`	128 * `PAPI_L3_DCM`	`PM_VECTOR_FLOP_CMPL`
	PGI 18.10	`-fast -Munroll`			
	IBM XL 16.1.1.2	`-qarch=pwr9 -qtune=pwr9`			

bytes exchanged with the main memory in that burst, t, the duration of the burst itself. This way for each phase p we can compute:

$$f_p = \sum_{j=1}^{P}\sum_{i=1}^{B} f_{i,j} \qquad m_p = \sum_{j=1}^{P}\sum_{i=1}^{B} m_{i,j} \qquad t_p = \max_{j=1}^{P}\sum_{i=1}^{B} t_{i,j} \tag{4}$$

Since the architectures under study offer different sets of hardware counters, we measure f and m using the PAPI events as described in Table 3.

Since the ThunderX2 CPU powering Dibona does not expose a counter of floating point operations, we approximated it from `PAPI_FP_INS` and `PAPI_VEC_INS`. Our approximation holds under the following assumptions:

- `PAPI_*_INS` count retired instructions. The counters do not include instructions issued speculatively and then squashed.
- Each instruction performs a single operation per floating point element.
- The vector instructions always use the whole 128 bits NEON register.

Also, in the ThunderX2 CPU, the PAPI version we are using does not support any last level cache counter. Instead, we use the L2 data cache miss counter, aware that it overestimates the traffic to the main memory.

4.2 Sustained performance

In this subsection, we use the metrics just introduced, f, m, and t to place the phases of Alya within the roofline model. Figure 4 plots the measured performance in each phase under the roofline curve of each cluster. The x-axis represents the arithmetic intensity $I = f/m$ (in Flops/Byte) and the y-axis represents the computational performance $F = f/t$ (in GFlop/s). The three machines and each point represents the pair (I_p, F_p) for each phase p of Alya on a given cluster using a specific compiler. Hence, the distance from each point to the roofline curve in Figure 4 represents a theoretical room of improvement (please note that both axes are represented in a logarithmic scale).

First of all, we show how the arithmetic intensity varies drastically depending on the architecture due to the different micro-architectures of the memory hierarchies. The point with the highest arithmetic intensity of each combination of machine and compiler corresponds to the element assembly phase. While in the solver and in the boundary operations all compilers in all architectures deliver similar arithmetic intensity and similar computational performance (points are mostly overlapping in Figure 4), in the case of the element assembly we see that points of different compilers are slightly scattered. We notice in particular that the GNU compiler

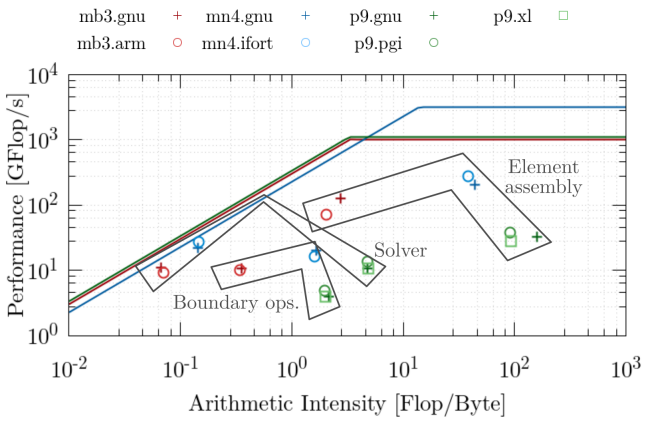

Figure 4: Roofline model of Dibona, MareNostrum4 and Power9 and measured performance per phase

enables a better use of the memory hierarchies (higher arithmetic intensity: +35% on Dibona, +15% on MareNostrum4 and +72%).

Secondly, the boundary operations appears to be the phase falling furthest away from the theoretical peak on all architectures. The reason comes from the nature of this phase. Since it deals with boundary elements that heavily vary with the geometry of the input, it has been less optimized for any specific architecture.

Lastly, Alya is not optimized for a specific architecture. However, Figure 4 shows that MareNostrum4 is systematically the closest to the peak. We consider this as a natural consequence of the higher maturity of the x86 HPC ecosystem.

4.3 Instructions Per Cycle

The IPC is a relevant metric to understand how busy is a CPU, so in an HPC context, where usually only one application per CPU is running, it is a good performance indicator for a given application. Figure 5 shows a plot that correlates the average elapsed time of the element assembly phase (x-axis) with the IPC during the same phase (y-axis). For our study, we use five compute nodes of each of the clusters, and we bind one MPI process per core. Therefore, we have a different number of MPI processes per cluster: 240 in MareNostrum4, 320 in Dibona, and 200 in Power9. Each point in Figure 5 represents the average value per MPI process, and the lines depict the standard deviation. A wide horizontal line means a large variability in execution time, while a tall vertical line means a big variability in IPC across processes. The top-left corner represents a short execution time with a high IPC, while the bottom-right corner represents a long execution time with a lower IPC.

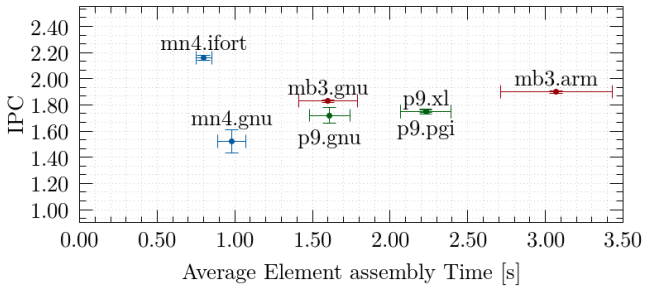

Figure 5: IPC and average duration of the element assembly

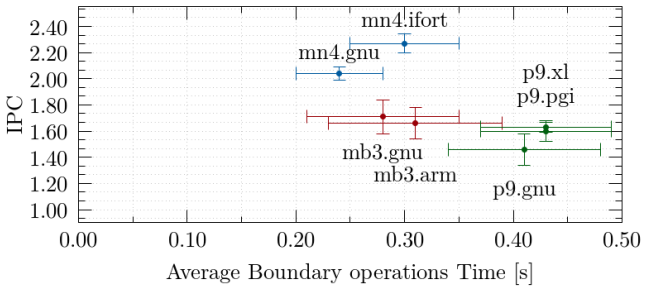

Figure 7: IPC and average duration of the boundary ops.

If we compare the element assembly time on each machine, it is clear that MareNostrum4 has a lower elapsed time. The Intel Compiler achieves a 2× speedup with respect to the best cases of Dibona and Power9. It is interesting to note that the GNU compiler on Dibona produces a binary with a higher IPC than in MareNostrum4. Since MareNostrum4 has only a 5% higher clock speed, this points at that the longer execution time in Dibona is due to a higher number of instructions being executed.

Comparing compilers within the same machine, we note that the GNU compiler produces a binary delivering more performance than the vendor-specific compilers on Dibona and Power9. We do not have a clear reason for this.

Figure 6 shows the same plot for the solver phase. In this case, all durations on the x-axis are within 0.20 and 0.35 seconds, regardless of the machine and compiler. It is important to note than, on Dibona, there is a higher variability than in the rest of the machines.

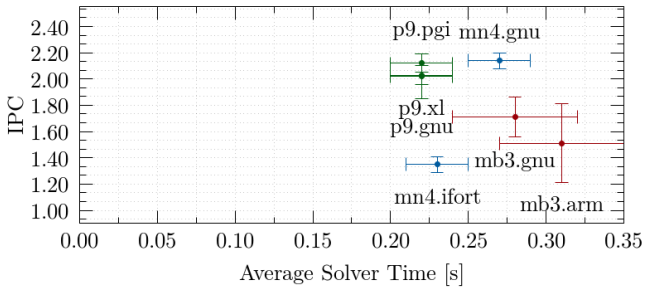

Figure 6: IPC and average duration of the solver

Figure 7 shows the same plot for the boundary operations phase. This phase is similar to the solver in that there seems to be little difference between machines and compilers even if there is a higher variability on elapsed time in Power9.

All in all, the element assembly phase, which represents the more significant part of the time step, is also the phase with a higher difference in performance across machines.

4.4 Vectorization

In Figures 5, 7, and 6 the reader should note that the IPC can significantly change when running a binary generated with different compilers for the same architecture. A change of IPC on the same cluster that is running at a given frequency can be due to either a different number of instructions generated by the compiler or the use of different types of instructions that implies different latencies.

A typical case that can generate such differences in IPC values happens when a compiler can generate more SIMD instructions than another. The difference in the *autovectorization* performed by the compiler and the width of the SIMD unit in each machine could indeed increase/decrease the total executed instructions. For this reason, we decided to study the vectorization of the different compilers available on the HPC clusters under evaluation.

Each machine has different performance counters to count vector instructions, but there is no common counter for all of them. Table 3 shows the PAPI counters we used to measure compiler autovectorization on each machine and their corresponding description.

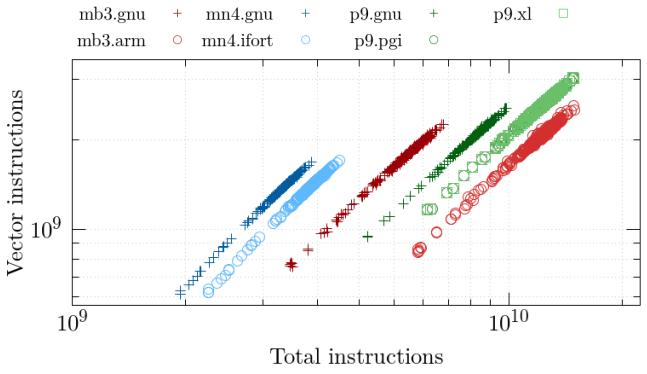

Figure 8: Compiler autovectorization in element assembly

Figure 8 shows the relationship between the total executed instructions (x-axis) and the vector instructions (y-axis) on the element assembly phase. Each point represents the measurement of one MPI process – points of the same color and type form a cluster. If a cluster spreads in the x direction, it means that the MPI processes are affected by load imbalance as defined in Section 2.

Our measurements show that the binaries with a lower elapsed time in Figure 5 are also the ones with a smaller number of executed instructions. In the case of MareNostrum4, there is little difference between the GNU and the Intel Compiler. For Dibona, the binary generated with the GNU compiler v8 executes half the number of instructions than the one generated with the Arm HPC Compiler. This may be the reason why the binary generated with the Arm HPC Compiler takes twice as long in this phase.

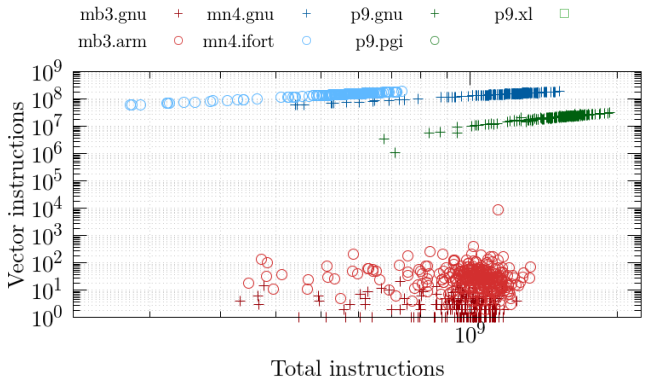

Figure 9: Compiler autovectorization in solver

Figure 9 shows the same measurements for the solver phase. The compilers in Dibona are not able to exploit the vector unit in this phase. Neither do the PGI and XL compilers in Power9, which generate zero vector instructions and do not appear in the plot.

It is also interesting to note that the solver phase from the roofline model in Figure 4 appears to be memory bound, so it should run faster on a platform with higher memory bandwidth like Dibona. However, Figure 6 tells us that Dibona is the slowest in executing this phase. The reason that makes MareNostrum4 outperforming Dibona is that the solver phase also includes arithmetic instructions. So the wider SIMD unit of MareNostrum4 and the better autovectorization achieved by the compiler allow overcoming the memory bandwidth limitations.

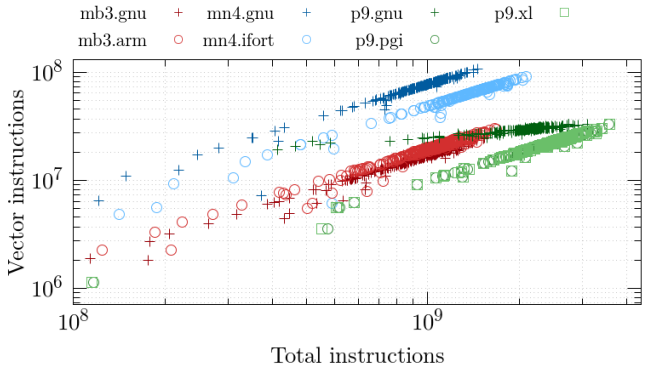

Figure 10: Compiler autovectorization in boundary ops.

Figure 10 shows the measurements in the boundary operations phase. In Figure 7 we showed that executions in Power9 were slower and we now show that a higher number of executed instructions may be the cause.

5 SIMULTANEOUS MULTI-THREADING ANALYSIS

All CPUs powering the clusters included in our study offer the possibility of enabling Simultaneous Multi-Threading (SMT). We evaluate this hardware feature under the HPC workload of Alya. All data presented in this section are measured averaging 19 time steps of Alya using 5 nodes of each cluster, and the error bars depict the standard deviation. For Dibona and MareNostrum4 tests with one SMT have been performed disabling SMT at boot. All other tests use nodes in which SMT has been set to the maximum number of SMT available at boot time.

Figure 11 shows the elapsed time by one time step when using a different number of SMTs per core in each cluster. The y-axis represents execution time (lower is better). Our results show no apparent performance benefit in using more than one hardware thread per core regardless of the machine and the compiler. In fact, for some compilers, the elapsed time increases drastically. On Dibona when using GNU, for instance, the time step duration increases ~10× when using three SMTs per core respect to using two SMTs per core.

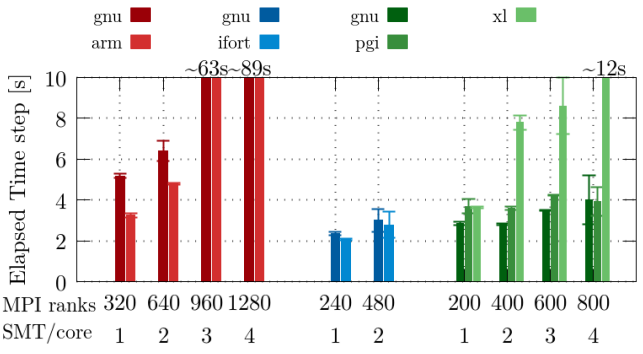

Figure 11: Time step elapsed time when increasing SMT per core

Looking at the elapsed time separated by phase delivers more insight on where the performance is lost. In Figure 12, we plot the elapsed time in the element assembly phase. We can observe that for all the clusters and all the compilers (except XL on Power9, which we discuss separately) the performance using two SMTs per core is the same as using one SMT per core. In some cases, a small benefit can be obtained, but it can not be considered significant if we look at the error bars.

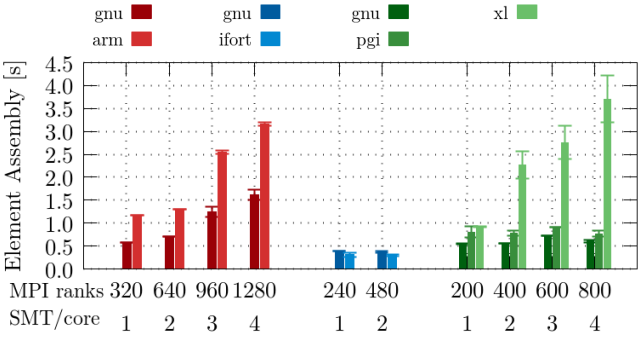

Figure 12: Element assembly elapsed time when increasing SMT per core

Using more than two SMTs per core (i.e., three and four) gives a worse performance in Dibona, independently of the compiler,

compared to using one or two SMTs per core. On the other hand, for Power9 when using GNU or PGI, the performance using three or four SMTs per core is comparable to the one obtained using one or two SMTs per core. The difference between the two machines can come from architectural differences but also from a saturation of shared resources. In the assembly phase, the compilers that vectorize "better" (e.g., GNU and Intel in MN4) are the ones less penalized by the use of SMT while the ones that vectorize less the code are more penalized (e.g., Arm in Dibona).

We studied the performance of the XL compiler in the assembly phase further due to the significant difference with respect to the other results. We observe that there seems to be a frequency scaling in the nodes during the assembly phase when using XL with several SMTs per core (we see low values of cycles per microsecond only in this phase). To obtain a more concluding remark, we would need more insight on the code generated by the compiler and the behavior of the node. Therefore, we leave it as future work in collaboration with the vendor.

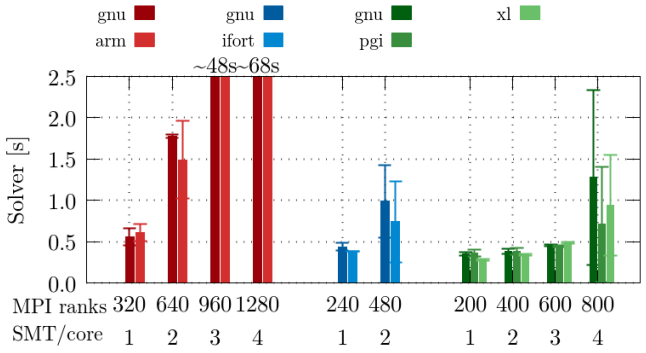

Figure 13: Solver elapsed time when increasing SMT per core

In Figure 13, we report the elapsed time of the solver phase. We can see that the performance obtained by using a different number of SMTs per core is highly dependent on the architecture and the compiler. In Dibona when running three and four SMTs per core, the solver phase accounts for ~76% of the total time step elapsed time. We know that the solver has a high number of communications, so the network will be more stressed in Dibona due to the higher number of processes per node (having Dibona the largest number of cores per node, e.g., each node has 256 MPI processes when using 4 SMTs).

In Figure 14, we plot the execution time of the boundary operations phase in the different clusters and with the different compilers. The boundary operations phase is the only phase that can benefit from SMT but only up to two SMTs in Dibona and MareNostrum4. This can be because this is the phase with a lower arithmetic intensity and it does not saturate the memory bandwidth. Also, using three or four SMTs in Dibona is ~5× slower than using one or two SMTs per core. In the case of Power9, the compilers do not have a high impact on the performance, having the three of them a similar trend, i.e., the best configuration using one SMT per core, and the worst one using three SMTs per core.

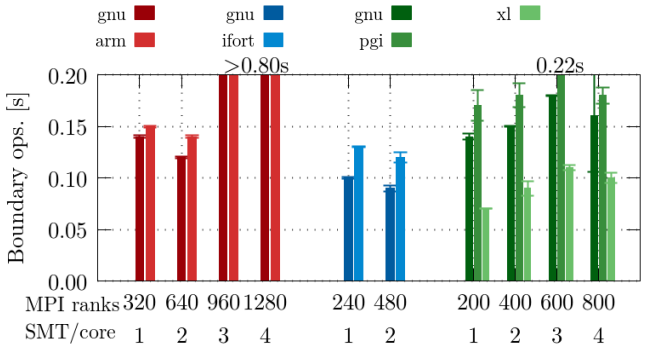

Figure 14: Boundary ops. elapsed time when increasing SMT per core

6 SCALABILITY

In this section, we present two scalability studies, the first one up to 32 nodes across all HPC clusters, and the second one up to 256 nodes only in MareNostrum4. All results presented in this section are "by node", although the different clusters have a different number of cores per node. Also, all data are measured averaging 19 time steps of Alya, and the error bars depict the standard deviation. We use GNU for all the clusters and one SMT per core in all the runs.

6.1 Scalability airplane 31.5 million elements

The experiments presented here are obtained using the same input as in the previous sections.

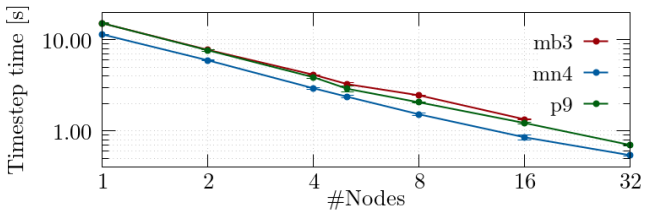

Figure 15: Time step scalability in all clusters

In Figure 15, we can see the scalability of a time step in the three clusters. We can observe that Dibona and Power9 perform similarly until four nodes. When using more than four nodes, the execution time in Dibona becomes slightly longer than in Power9.

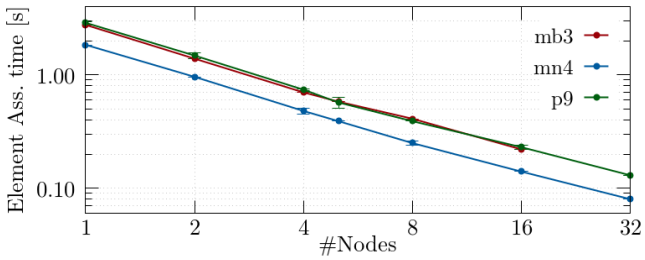

Figure 16: Element assembly scalability in all clusters

In Figure 16, we plot the elapsed time in the element assembly phase. In this phase, Dibona and Power9 have the same performance per node up to 16 nodes. The three clusters present a scalability of the element assembly phase close to the ideal.

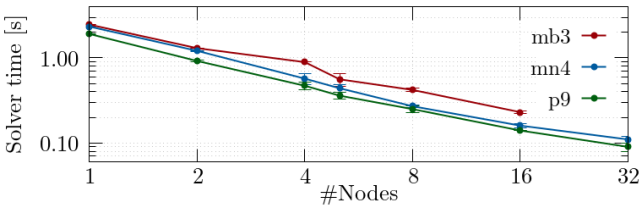

Figure 17: Solver scalability in all clusters

In Figure 17, we present the execution time of the solver phase. We show that the cluster that delivers the best performance is Power9, outperforming MareNostrum4 by a 20%. This is coherent with observations of Figures 6 and 13, that the solver presents lower execution times in Power9 than in MareNostrum4 (also 20% lower). This can be explained by the frequency at which Power9 cores operate (3.0 GHz) compared to the MareNostrum4 one (2.0 GHz). Dibona performs worse than the other two clusters in the solver. This was already shown in Figure 6, we concluded that GNU is not able to generate a binary that exploits the vector units of the core.

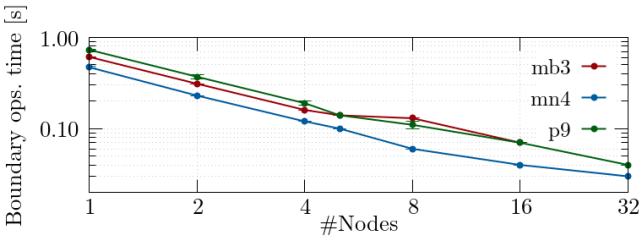

Figure 18: Boundary ops. scalability in all clusters

Finally, in Figure 18, we can see the elapsed time in the boundary operations phase. We can observe that in this phase, MareNostrum4 is the one with the lowest elapsed time, but the efficiency degrades when scaling beyond eight nodes. This phase, as explained in Section 2.2, is highly dependent on the partition (and consequently on the number of MPI processes). The Load Balance[3] of the boundaries operations with 4 nodes in MareNostrum4 is 0.83 while when running on 16 nodes it is 0.72.

Dibona performs better than Power9 in this phase up to 4 nodes as was expected based on the results from previous sections (see e.g., Figures 7 and 14).

6.2 Scalability airplane 252 million elements

Finally, in this subsection, we present the scalability up to 256 nodes in MareNostrum4. For this study, we use a more detailed mesh of the same airplane; the input mesh used has 252 million elements.

In Figure 19, we show the scalability of a single time step when simulating the 252 million elements airplane in MareNostrum4.

[3]The Load Balance is computed as explained in [7], section IV-i, equation 12

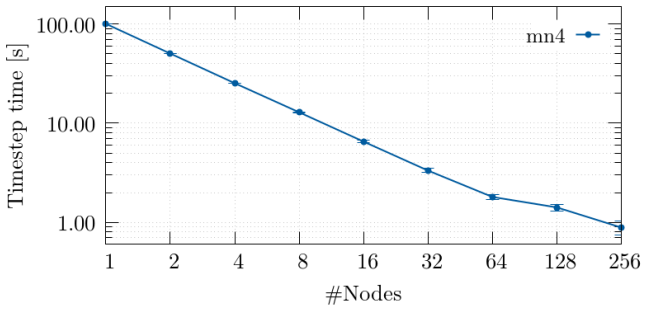

Figure 19: Time step scalability in MareNostrum4

We can see that the whole simulation scales well up to 64 nodes (3072 cores). However, when using 128 and 256 nodes, the parallel efficiency drops. It should be noted that in these two cases, the workload per MPI process is very low (40 and 20 thousand elements, respectively).

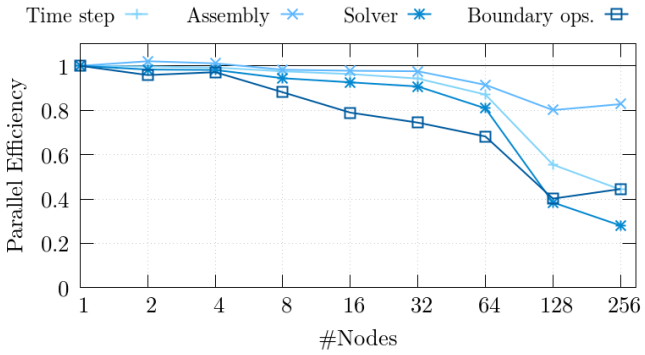

Figure 20: Parallel efficiency up to 256 nodes

In Figure 20, we present the parallel efficiency of each phase. The efficiency E has been computed as follows: $E = t_1/(i \cdot t_i)$, where t_1 is the execution time when running with one node and t_i is the execution time when running with i nodes. Therefore, efficiency will take values in the [0..1] range, being 1 the ideal parallel efficiency. We observe that the phases responsible for the loss of performance for more than 64 nodes are the algebraic solver and the boundaries operations. In particular, the performance of the solver drops for 128 and 256 nodes. The boundary operations phase also has a poor parallel efficiency, but it steadily drops between 4 and 128 nodes. Also, the parallel efficiency of the element assembly phase is good up to 256 nodes.

In Figure 21, we can see the percentage of the time step time spent in each of the phases when increasing the number of nodes. We can see that the assembly phase is the dominant one up to 64 nodes. For more nodes, the solver becomes the dominant phase and also the bottleneck for the scalability, as we have seen in Figure 20.

In Figure 22, we show the percentage of time step spent in MPI communication when increasing the number of MPI ranks. We can observe that the time in MPI increases drastically with the number of nodes and MPI processes used. Therefore, we can say that the

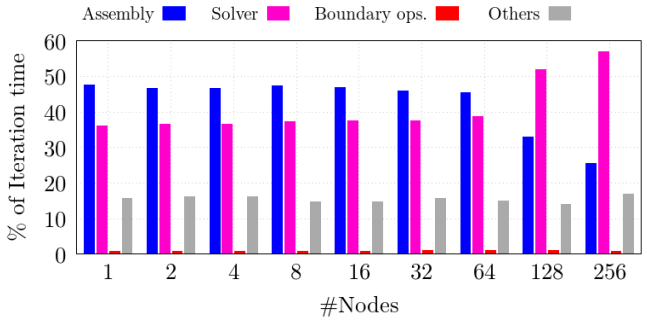

Figure 21: Percentage of time spent in each phase

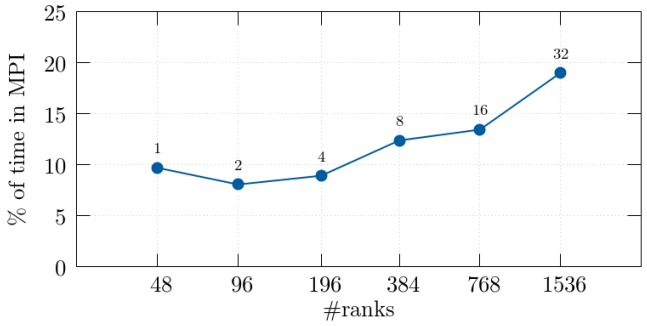

Figure 22: Percentage of time in MPI up to 32 nodes

performance loss observed in Figure 20 is due to MPI communication. Nevertheless, with the current data, we cannot demonstrate if the performance loss of MPI is due to the load balance, the overhead of MPI, or the transfer time of the network.

7 CONCLUSIONS

We studied a production CFD application on three state-of-the-art clusters powered by different CPU architectures.

On Dibona, both vendor and GNU compilers are not able to autovectorize the solver. More extreme is the case of PGI and XL compilers on the Power9 cluster: we showed in fact that they do not vectorize at all the code of some of the phases of Alya. On the other hand, the Intel compiler on MareNostrum4 seems to be the one delivering better autovectorization. The two lessons learned are: on the one hand, the maturity of both architectures and compilers delivers better performance, on the other hand, that different compilers for a given architecture can extract a very different level of autovectorization from the same code.

The performance of Simultaneous Multi-Threading (SMT) highly depends on the architecture and the compiler. In the use case at study, we do not detect significant benefits in using more than one SMT per core. However, our study highlights that the impact of SMT varies among different phases. Therefore it is not possible to choose a single configuration that benefits all codes.

The scalability of the application on a high number of cores is limited by the phase that has more MPI communication and load imbalance at MPI level. Although, these were not the phases with more impact in low core counts.

Complex HPC applications such as Alya are structured in different phases. Figure 16 shows that the cluster performing better for the element assembly phase is MareNostrum4, while Figure 16 shows that Power9 is the most suitable cluster for computing the solver phase. Once more, the outcome of our study is twofold. On the one hand, when dealing with complex applications, there is no unique recipe that can maximize the performance of all phases. On the other hand, we verified that two factors that always improve the performance are a wide SIMD unit coupled with a compiler able to autovectorize and a high-frequency CPU.

ACKNOWLEDGMENTS

The authors thank the support team of Dibona operating at ATOS/Bull. This work is partially supported by the Spanish Government through Programa Severo Ochoa (SEV-2015-0493), the Spanish Ministry of Science and Technology (project TIN2015-65316-P), the Generalitat de Catalunya (2017-SGR-1414), the European project Mont-Blanc 3 (GA n. 671697), and the European centre of excellence on HPC POP2 (GA n. 824080).

REFERENCES

[1] Fabio Banchelli, Marta Garcia-Gasulla, Marc Josep-Farego, et al. 2019. *MB3 D6.9 – Performance analysis of applications and mini-applications and benchmarking on the project test platforms.* Technical Report. http://bit.ly/mb3-dibona-apps
[2] Claudio Bonati et al. 2018. Early Experience on Running OpenStaPLE on DAVIDE. In *High Performance Computing (Lecture Notes in Computer Science).* Springer International Publishing, 387–401.
[3] Enrico Calore, Filippo Mantovani, and Daniel Ruiz. 2018. Advanced performance analysis of HPC workloads on Cavium ThunderX. In *2018 International Conference on High Performance Computing & Simulation (HPCS).* IEEE, 375–382.
[4] Sergey Charnyi et al. 2017. On conservation laws of Navier-Stokes Galerkin discretizations . *J. Comput. Phys.* 337 (2017), 289 – 308.
[5] Ramon Codina. 2001. Pressure stability in fractional step finite element methods for incompressible flows. *J. Comput. Phys.* 170 (2001), 112–140.
[6] Jean Donea and Antonio Huerta. 2003. *Finite Element Methods for Flow Problems.* John Wiley & Sons. https://books.google.es/books?id=S4URqrTtSXoC
[7] Marta Garcia-Gasulla, Filippo Mantovani, Marc Josep-Fabrego, Beatriz Eguzkitza, and Guillaume Houzeaux. 2019. Runtime mechanisms to survive new HPC architectures: A use case in human respiratory simulations. *The International Journal of High Performance Computing Applications* (2019).
[8] James A Kahle, Jaime Moreno, and Dan Dreps. 2019. 2.1 Summit and Sierra: Designing AI/HPC Supercomputers. In *2019 IEEE International Solid-State Circuits Conference-(ISSCC).* IEEE, 42–43.
[9] Damian Kaliszan, Norbert Meyer, and Sebastian others Petruczynik. 2019. HPC Processors Benchmarking Assessment for Global System Science Applications. *Supercomputing Frontiers and Innovations* 6, 2 (2019), 12–28.
[10] Karypis George and Kumar Vipin. 2009. *MeTis: Unstructured Graph Partitioning and Sparse Matrix Ordering System.* Technical Report. University of Minnesota, Minneapolis.
[11] James H Laros, Kevin Pedretti, Simon David Hammond, et al. 2018. *FY18 L2 Milestone# 8759 Report: Vanguard Astra and ATSE? an ARM-based Advanced Architecture Prototype System and Software Environment.* Technical Report. Sandia National Lab.(SNL-NM), Albuquerque, NM (United States).
[12] Oriol Lehmkuhl, Guillaume Houzeaux, Herbert Owen, Giorgios Chrysokentis, and I Rodriguez. 2019. A low-dissipation finite element scheme for scale resolving simulations of turbulent flows. *J. Comput. Phys.* (2019). Accepted.
[13] Rainald Löhner. 2008. *Applied Computational Fluid Dynamics Techniques: An Introduction Based on Finite Element Methods.* John Wiley & Sons.
[14] Rainald Löhner and Joseph D. Baum. 2014. On maximum achievable speeds for field solvers. *International Journal of Numerical Methods for Heat & Fluid Flow* 24, 7 (2014), 1537–1544. https://doi.org/10.1108/HFF-01-2013-0016
[15] Matt Martineau and Simon McIntosh-Smith. 2017. Exploring On-Node Parallelism with Neutral, a Monte Carlo Neutral Particle Transport Mini-App. In *2017 IEEE International Conference on Cluster Computing (CLUSTER).* 498–508. https://doi.org/10.1109/CLUSTER.2017.83
[16] Simon McIntosh-Smith. 2017. A Survey of Application Memory Usage on a National Supercomputer: An Analysis of Memory Requirements on ARCHER. In *High Performance Computing Systems. Performance Modeling, Benchmarking, and Simulation: 8th International Workshop, PMBS 2017, Denver, CO, USA, November 13, 2017, Proceedings*, Vol. 10724. Springer, 250.

[17] Simon McIntosh-Smith, James Price, Tom Deakin, and Andrei Poenaru. 2018. A performance analysis of the first generation of HPC-optimized Arm processors. *Concurrency and Computation: Practice and Experience* (2018), e5110.

[18] Georg Ofenbeck, Ruedi Steinmann, Victoria Caparros, et al. 2014. Applying the roofline model. In *2014 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, 76–85.

[19] Herbert Owen, Georgios Chrysokentis, et al. 2020. Wall-modeled large-eddy simulation in a finite element framework. *International Journal for Numerical Methods in Fluids* 92, 1 (2020), 20–37.

[20] Vincent Pillet, Jesús Labarta, Toni Cortes, and Sergi Girona. 1995. Paraver: A tool to visualize and analyze parallel code. In *Proceedings of WoTUG-18: transputer and occam developments*, Vol. 44. IOS Press, 17–31.

[21] Stephen B Pope. 2011. *Turbulent flows*. Cambridge Univ. Press, Cambridge. https://cds.cern.ch/record/1346971

[22] Nikola Rajovic, Alejandro Rico, Filippo Mantovani, et al. 2016. The Mont-Blanc prototype: An alternative approach for HPC systems. In *SC'16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 444–455.

[23] Junuthula Narasimha Reddy. 2005. *An Introduction to the Finite Element Method*. McGraw-Hill Education. https://books.google.es/books?id=8gqnRwAACAAJ

[24] Daniel Ruiz, Filippo Spiga, Marc Casas, Marta Garcia-Gasulla, and Filippo Mantovani. 2019 in press. Open-Source Shared Memory Implementation of the HPCG Benchmark: Analysis, Improvements and Evaluation on Cavium ThunderX2. In *2019 International Conference on High Performance Computing & Simulation (HPCS)*. IEEE. http://hdl.handle.net/2117/116642

[25] Yousef Saad. 2003. *Iterative Methods for Sparse Linear Systems*. SIAM.

[26] Harald Servat, Germán Llort, Kevin Huck, Judit Giménez, and Jesús Labarta. 2013. Framework for a productive performance optimization. *Parallel Comput.* 39, 8 (2013), 336–353.

[27] Sudharshan S. Vazhkudai, Bronis R. de Supinski, Arthur S. Bland, et al. 2018. The Design, Deployment, and Evaluation of the CORAL Pre-exascale Systems. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis (SC '18)*. IEEE Press, 52:1–52:12.

[28] Mariano Vázquez, Guillaume Houzeaux, Seid Koric, et al. 2016. Alya: Multiphysics Engineering Simulation Toward Exascale. *Journal of Computational Science* 14 (2016), 15–27.

[29] A.W. Vreman. 2004. An eddy-viscosity subgrid-scale model for turbulent shear flow: Algebraic theory and applications. *Physics of Fluids* 16, 10 (2004), 3670–3681.

[30] Samuel Williams, Andrew Waterman, and David Patterson. 2009. *Roofline: An insightful visual performance model for floating-point programs and multicore architectures*. Technical Report. Lawrence Berkeley National Lab. (LBNL).

[31] Yuzuru Yokokawa et al. 2006. Experimental and CFD of a High- Lift Configuration Civil Transport Aircraft Model. *AIAA Paper 2006-3452* (2006).

[32] Yuzuru Yokokawa, Mitsuhiro Murayama, et al. 2007. Investigation and Improvement of High-lift Aerodynamic Performances in Low speed Wind Tunnel Testing. *AIAA Paper 2008-350* (2007).