Universitat Politècnica de Catalunya (UPC) - BarcelonaTech

Facultat d'Informàtica de Barcelona (FIB)

Master in Innovation and Research in Informatics

High-Performance Computing

Methods and measurements for evaluating HPC systems

Author:
Fabio Banchelli

Advisor:
Filippo Mantovani

Tutor:
Jesus Labarta

Co-advisor:
Marta Garcia-Gasulla

Department:
Computer Architecture

Company:
Barcelona Supercomputing Center

June 2020

Abstract

In this thesis, I present a methodology to evaluate High-Performance Computing systems. The method relies on measurements at three levels: architectural features through micro-benchmarks; system software and tools through benchmarks and analysis of the code generated by the compiler; and sustained performance though scientific applications. I apply the method to three state-of-the-art High-Performance Computing clusters deployed at the Barcelona Supercomputing Center.

In dedication to:

Claudio, who taught me to be ***persistent***
Isabel, who taught me to be ***methodical***
Toya and Paco, who taught me to be ***curious***

Contents

Chapter 1

Introduction

This thesis is the result of the work I contributed to in the "Mobile and embedded-based HPC" group within the Barcelona Supercomputing Center (BSC). In the context of this Master's thesis, I contributed to the following peer reviewed publications:

- Corresponding author and speaker of the research poster [1].
- Contributor in the deliverable [2].
- Contributor in the journal article [3] (in press).
- Contributor in the paper [4].
- Corresponding author and speaker of the paper [5] (I was not able to present the work due to the outbreak).

I construct an evaluation methodology which can be applied to High-Performance Computing (HPC) systems based on different Central Processing Unit (CPU) arhcitectures. This work originates from the Mont-Blanc European project, which needed to evaluate multiple HPC systems based on different architecture under a certain time constraint.

In this thesis I apply my method for the evaluation of three different HPC clusters powered by different CPU architectures: Dibona, an Arm-based cluster, Power9, an cluster housing IBM-Power9 CPUs and MareNostrum4, the Intel-based flagship supercomputer of BSC. I structured my evaluation with a bottom-up approach, executing programs with an increasing level of complexity. While I use simple kernels/benchmarks for measuring architectural features of each cluster, I use a complex Computational Fulid Dynamics (CFD) application to validate my method with a real scientific code. The methodology I present in this thesis is based on theoretical performance models that can be built from empirical experiments. For each one of the experiments, I try to not only present the results, but also discuss their causes and repercussions. I found out that it is possible to generalize a methodology so that it works across multiple machines. Even when only evaluating the CPU of a system, there are micro-architectural features and software differences that make it difficult to keep the methodology consistent for all CPU architectures.

The rest of the document is structured as follows. Chapter 2 gives a broad overview of the HPC landscape, covering some of the technological challenges that the industry is currently facing and discussing methods to evaluate HPC clusters that are available in the literature. Chapter 3 details the technical knowledge on which this thesis is built upon. I start by briefly explaining computer architecture concepts to understand how HPC clusters work and presenting three HPC clusters which I evaluate in this thesis. I follow up introducing the two performance evaluation models, the Roofline model and the Performance Optimization and Productivity (POP) Efficiency model, I use to characterize the three machines. In Chapter 4, I evaluate the system as close to a bare-metal environment as possible. The experiments I present in this chapter are simple codes that aim to stress a particular component of the HPC system. With the experiments I perform in this chapter, I construct the Roofline model, which characterizes a machine in terms of its attainable performance. Chapter 5 evaluates the compilers and communication libraries available on the three machines. Chapter 6 presents a performance study of a production scientific application leveraging an Efficiency model developed at BSC. Chapter 7 wraps up the thesis by discussing the conclusions and lessons learned.

Chapter 2

Context

Cuando creíamos que teníamos todas las respuestas,
de pronto, cambiaron todas las preguntas.

"Just when we thought we had all the answers,
suddenly, all the questions changed."

— Mario Benedetti

2.1 Technological challenges

The decade of the 2020 will be the finish line of the race towards exascale. The new generation of top HPC systems will be able to compute 10^{18} Floating-point operations per second (Flop/s). China expects to have the Tianhe-3 supercomputer up and running by the end of 2020 [6]. Argonne National Laboratory (ANL) announced in 2019 that their exascale system, named Aurora, will be operational in 2021 [7]. Oak Ridge National Laboratory (ORNL) will also have an exascale system by 2021 named Frontier [8]. The European Union targets 2022/2023 for its own exascale system with joint undertakings as the EuroHPC [9].

The road to exascale has brought a great diversity of CPU architectures to HPC. The current fastest supercomputer, Summit, is based on IBM's Power9 architecture. ANL's Aurora will use Intel's x86 architecture while ORNL's Frontier will feature the newest CPUs from AMD. On the other hand, is expected that the Tianhe-3 will feature some kind of Arm CPU. The European Union has also hinted at using Arm while also developing its own technology based on the RISC-V architecture during the European Processor Initiative (EPI) [10]. Given the sudden diversity in CPU architectures, some call this period the *Cambrian explosion* of High-Performance Computing.

Going beyond raw performance

But what does an exascale system entail? Up until recently, we mainly focused on raw performance: Flop/s. The Top500 list [11] updates biannually with the newest and most powerful systems. The rankings are determined by the sustained performance using a single benchmark: High-Performance Linpack (HPL) [12, 13]. In recent years, people have come to realize that HPL is a benchmark that mirrors a very specific computing task (i.e., solving dense linear algebra systems). This benchmark has been designed to stress the floating-point units of the system. In a sense, HPL represents a best-case-scenario where the system is computing at almost its full capacity.

Some argue that ranking HPC systems using HPL is not fair. To this end, a sibling benchmark has become a sibling to HPL: High-Performance Conjugate Gradient (HPCG) [14]. This benchmark represents almost the polar opposite to HPL, being a worst-case-scenario where the system is struggling to feed the floating-point units because is continuously fetching data. Even if HPCG is not considered for ranking in the Top500 list, there is a list of Top-HPCG scores. As an example, in the November 2019 list, Summit scores 148.60 PFlop/s with HPL and 2.93 PFlop/s with HPCG.

Historically, the HPC industry has developed CPUs that could rank higher in the Top500 rank. To achieve this goal, we crammed more and more floating-point resources into the CPU (e.g., more floating-point units, longer Single-Instruction Multiple-Data (SIMD) vectors, etc.). Results from the HPCG benchmark show that we may need to focus on other aspects in the design to achieve better performance. Domke et al. show in [15] how having more floating-point resources might not lead to higher performance.

Other challenges in the road to exascale

If performance is the driving force in computing, power dissipation is one of its most limiting factors. Summit, consumes 10 MW. On average, it can perform 14.72 GFlop per Watt. We call this ratio the power efficiency. If we were to project Summit's performance up to an exascale system maintaining its current power efficiency, we would need $\sim$ 70 MW to power the machine. Clearly, HPC systems have to improve its power efficiency alongside raw performance to reach sustainable EFlop/s.

In addition to the Top500, which measures performance, there is also the Green500 list [16], which ranks systems by its power efficiency. Looking at the November 2019 rank, we find that Summit, number one in the Top500, is number five in the Green500. The number one in the Green500, the A64FX prototype by Fujitsu [17], ranks 159 in the Top500. This system has a power efficiency of 16.88 GFlop/W. With the same efficiency, the system would need $\sim$ 59 MW to reach one EFlop/s. US DARPA published a report in 2008 which stated that 20 MW would be a reasonable power dissipation for an exascale system[18]. This means a minimum power efficiency of 50 GFlop/W. Looking at the November 2019 Green500 list, HPC systems have still a long way to go.

There are other roadblocks in the journey towards exascale computing. Heldens et al. in [19] give a broad overview of the landscape in HPC research up until 2020. Issues such as data management, resiliency and programmability are hot topics in the field of HPC.

Industry, academia and education

In the midst of the exascale race, vendors release new CPU models continuously. Intel's Skylake will eventually lead into Icelake, even though Intel has been struggling to keep up the pace. AMD recently released its Rome Epyc 7002 series [20]. Arm has been gaining traction in the HPC market[21, 22]. Arm does not produce chips itself, but sends its Intellectual Property (IP) instead. This business model allows other companies to develop their own chips based on the Arm architecture. Examples of this are Marvell with its ThunderX2 (TX2) and ThunderX3 server line; Fujitsu's A64FX; Huawei's Kunpeng 916; and Ampere's Quicksilver.

There is a broad selection of CPUs with different architectures reaching the HPC market at once. From the point of view of a company that uses HPC resources, it would be sensible to ask the question: What are the differences between all these CPUs? Which one performs better for the company's goals?

The same questions can be formulated in the context of academia. The Mont-Blanc European project [23] started in 2011 trying to leverage mobile technology for HPC [24]. By the end of 2018, the Mont-Blanc3 European project had a production cluster based on the Arm architecture [25]. From start to finish, the project developed multiple prototype systems which had to be evaluated in order to better understand how mobile technology benefited HPC.

The need for system evaluation also extends to the education environment in HPC. The Student Cluster Competition is an event where undergraduate students setup a mini-cluster and run HPC applications on a small time frame. Most students do not even know what High-Performance Computing is at the start of the competition, but they need to be able to leverage and evaluate a system within six months of preparation. Banchelli et al. explain in [26] the teaching method refined throughout multiple editions of the Student Cluster Competition.

In conclusion, industry, academia and education in HPC need a method to evaluate new CPU models under certain time constraints. This thesis studies how are HPC systems currently evaluated and tries to build a methodology which is applicable to different CPU architectures.

2.2 State of the art

There are multiple works in the literature where the authors evaluate HPC systems. In general, the evaluation of a machine is done at multiple levels, going from simple test up to complex applications. It is also common to have a baseline system to compare against. In [27], the authors evaluate a very recent Arm CPU. They focus first on measuring the floating-point performance and memory bandwidth. The authors chose HPL to measure raw performance. As I mentioned in Section 2.1, HPL is a benchmark that has gained some criticism because it does not represent a realistic HPC workload. To evaluate the systems' memory hierarchy, the authors use STREAM and lmbench. These are benchmarks commonly used in HPC to measure memory bandwidth and cache access latency. It is important to note that STREAM is composed of four distinct kernels, each one with a distinct memory access pattern. Most evaluations found in the literature report the Triad kernel, which implements a multiply-and-add operation over an array.

In [28], the authors evaluate a system based on a vector architecture. In recent years, vector architectures are having a resurgence in HPC (e.g., RISC-V "V" extension). Although the overall CPU design is very different to a general purpose core, the authors of [28] follow a similar approach to evaluating their system. They use STREAM to measure the memory bandwidth and some computational kernels to measure floating-point performance. The study ends with the performance evaluation of two scientific applications. I find interesting to note that, even for a pretty different CPU architecture, there is a similar evaluation to the one presented in [27].

There are multiple benchmarks and tools to evaluate different components of an HPC system. Some of them are presented as being portable to multiple CPU architectures. For benchmarks that aim to stress a single aspect of the CPU, it is difficult to take into account all the quirks and specifics of a given architecture (and/or micro-architecture) and keep the code portable. I personally find very interesting the work presented in [29], where the authors use a third-party tool to build and empirical performance model of a system. After doing some tests, the authors realized that the high level code of the computational part of the tool was written in such a way that made it so the benchmark could not fully leverage the computational resources of the CPU. To solve this issues, the authors had to manually modify the code, specializing it to fit the quirks of the system they were evaluating.

Contrary to popular opinion, the success or failure of a given architecture, micro-architecture or HPC system in general is not determined by its performance. The software ecosystem and tools user are much more critical to the final user. In 2019, Arm's senior director of the HPC business division, Brent Gorda, explained that the adoption of Arm in HPC was more than powerful chips and friendly IPs. Specifically, Gorda stated that *there is the need to provide software tools necessary for server-side application development.* The full interview is available at [21].

Following the idea of a strong software ecosystem, our group at BSC presented in [1] a preliminary evaluation of the Arm software ecosystem at the time. The study uses simple benchmarks to compare the GNU compiler toolchain and runtime libraries with the software packages provided by Arm.

In 2019, the authors in [30] presented an in-depth analysis of the Arm software ecosystem. Their work studies multiple compilers, compiler versions and optimization flags in a state-of-the-art Arm system. The key takeaway from this publication is that different compilers behave very differently; each one having their own options and flags. Moreover, flags with the same name across two compilers may have completely different meaning. Beyond the compiler, it is also important to consider the scientific libraries that complex applications need to use. The work of Jackson et al. shows that the implementation and configuration of these libraries also impact in the overall performance of a scientific application.

In [31], the authors evaluate the first Arm system to enter the Top500, Astra. Similarly, in [32] evaluate another Arm system, also based on the same CPU of Astra. In both publications, the evaluation methodology includes so called mini-apps (i.e., benchmarks) and scientific applications. For the applications, the authors compare the figure of merit of each application measured in the Arm system against another system that serves as a reference. In most cases, there is no further exploration on why the performance in one system is better or worse compared to the baseline. It is unclear if the origin of inefficiencies comes from the architecture, the software ecosystem or the code itself.

Lastly, in [33] the authors present the testing and evaluation of the fastest supercomputer in the Top500, Summit. Their work includes a description on how the system was deployed, which acceptance tests were performed and, finally, a discussions on the lessons learned. The authors explain that the work that they present took several months to accomplish.

Overall, I found out that most machine evaluations are organized in layers of complexity. The authors start by running kernels and micro-benchmarks that stress a particular component of the systems (e.g., STREAM and lmbench). In some cases, the study also includes some benchmarks that use some external libraries but are not as complex as full scientific applications (e.g., HPL and HPCG). Finally, the evaluation always includes execution of scientific applications. In most cases, the authors choose two or three applications. Some publications study a bigger set of applications. When this happens, the study tends to be limited to compare the figure of merit of the given application between to machines.

In this thesis, I present an evaluation methodology based on the work published in [2]. In this deliverable, the authors evaluate an HPC system at three levels: micro-benchmarks, benchmarks, and applications. A refined version of the study was published in [3]. In this publication, the authors include scalability analysis of four scientific applications. However, as with [31, 32], there is no analysis of why the applications might not scale. In this thesis, I incorporate in the evaluation methodology the performance model used in [4].

2.3 Research questions and contributions

Research questions

1. How can we precisely evaluate a new HPC cluster under a short time constraint?
2. When performing measurements with benchmarks or HPC applications, how can we spot inefficiencies and where do they come from?

Contributions

1. Definition of a methodology to evaluate HPC systems under the following constraints:
   - Provides insightful information (i.e., not a single figure of merit).
   - Is applicable to multiple CPU architectures.
   - Is applicable under sensible time constraints (e.g., the Student Cluster Competition).
2. Application of the methodology on three HPC clusters that have different CPU architectures.
3. Performance study of a production scientific application.

Chapter 3

Technical Background

Measure what is measurable, and make measurable what it is not so.

— Galileo Galilei

3.1 The architecture of a cluster

Core

The core is the basic computational unit of a cluster. In this thesis, I model the core as a simple machine that takes a sequence of instructions and executes them. We call *Instruction Set Architecture (ISA)*, or simply *architecture*, the set of executable instructions. Examples of ISAs are Intel's x86, Armv8 from Arm, and Power9 from IBM. Generally, architectures are incompatible between each other, meaning that a sequence of x86 instructions cannot be understood by an Armv8 core. However, common operations such as adding operands or storing data are present across all architectures, even if they are encoded differently.

We call *micro-architecture* the internal implementation of a core. The micro-architecture defines how a core executes instructions. Some micro-architectural features, like the number of stages of the execution pipeline, are available to the public. On the other hand, implementation details such as length of internal buffers are mostly undisclosed. An end user can only infer some of these details with empirical measurements. In this thesis, I consider the micro-architectural features of the core as a black box. I present methods to evaluate, empirically, the details of the micro-architecture implementation.

Even if we model the core implementation as a black box, we can identify some high level components that are present in all micro-architectures. For example, all general purpose cores have a specific unit to execute arithmetic operations. HPC applications are heavily based on floating point arithmetic, either with single or double precision. For this reason, micro-architectures targeting HPC have powerful Floating Point Unit (FPU) and SIMD units. The rate at which these floating point units execute instructions is called *Floating point throughput* and it is measured in Floating-point operations per second (Flop/s). Section 3.4 elaborates on this topic.

Figure 3.1 shows a simple schematic of how we model the core implementation. An instruction is fed to the control unit which triggers the execution. When this occurs, we say that the instruction has been issued. Source operands are read from the register bank and travel to the right into one of the multiple functional units of the core. For simplicity, I have only included two FPUs and one SIMD units, but there can be other types of units. Once the functional chosen unit has computed a result, it is stored in a reorder buffer, waiting until the control unit triggers a write operation to the register bank. When this occurs, we say that the instruction has been committed.

Please note that the model I present in this section and the schematic shown in Figure 3.1 are simplifications of a much more complex system. However, our simplified model is enough to carry on the methodology I present in this thesis.

As a concluding remark I would like to mention that some cores implement Simultaneous Multi-Threading (SMT), which effectively multiplexes the operational units of the core between multiple instruction threads. All the machines I study in this thesis implement SMT but it has been disabled for all experiments.

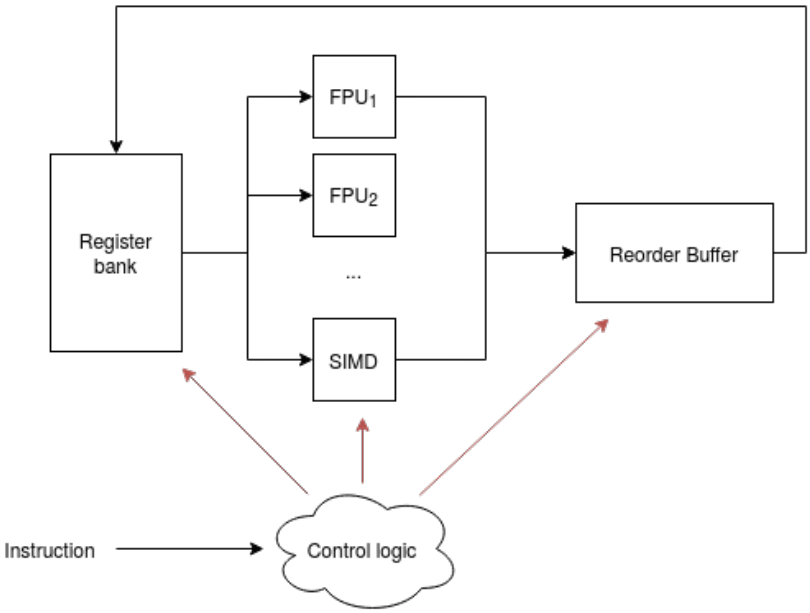

Figure 3.1: Simplified schematic of the functional units inside a core and how they are connected

Memory hierarchy

The memory is a unit that stores data. Historically, the memory has been divided into multiple levels, forming a hierarchy depending on the access speed to the stored data. The simplest memorization element is the register or register bank. Registers are physically inside the core and serve as a very quick access memory (in the order of a cycle). Most instructions have one or more registers as source operands and one register as a place to store the result of the operation.

The capacity of the register bank is very limited (in the order of KiB at most). The main memory, or simply memory, is the memorization element that sits on top of the register bank. The capacity of the memory is much greater than the register bank (in the order of GiB) at the cost of being much slower (access time in the order of hundreds of cycles). The main memory is physically outside of the core. Most ISAs include instructions to access the main memory. The rate at which data is transferred to/from memory is called *Memory bandwidth* and it is measured in Bytes per second (B/s). Section 3.4 elaborates on this topic.

All modern core implementations add levels to the memory hierarchy that serve as halfway points between the register bank and the main memory. These intermediate memorization units are called caches or cache levels. Cache levels are named following a numbering scheme: L1 cache for the smallest and closest to the core, followed by the L2 cache and so on. State-of-the-art processors use memory hierarchies with three levels of cache, having the first cache level inside the core. The behavior and implementation of caches is an active research field. In this thesis, I do not provide details on the cache implementation of each machine. This study is limited to measure the access time and bandwidth across the memorization elements of the memory hierarchy.

There is an additional memorization element above the main memory. We call this last level of the memory hierarchy the disk or external storage. Access time to the disk is in the order of millions of cycles but can store, virtually, an infinite amount of data. Most scientific applications access the disk to read the input set of a simulation and store the result back to the disk. Generally, the access to the disk is both at the beginning and end of the program execution, and it has no impact on the performance of the computational part of the simulation. Other workloads require a lot of data movement with the disk and have a very important impact on the execution time. For this reason, storage is an active research field and has been marked as a major challenge in the road to Exascale [19]. In this thesis I do not study the impact of the external storage.

Figure 3.2 shows a schematic view of the memory hierarchy. Memorization elements at the top have the least capacity and access time, while the ones at the bottom have the highest capacity and access time. The annotations at either sides of the pyramid show a rough estimation of the level of magnitude in capacity and access time for each level.

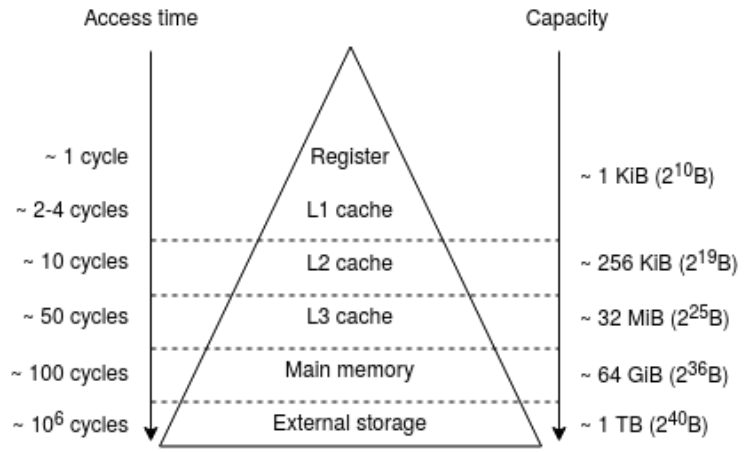

Figure 3.2: Classical depiction of the memory hierarchy

CPU and node

We call CPU the physical package sold by vendors. The CPU contains on or more cores. In fact, all modern CPUs are composed by multiple cores that work in parallel. The number of cores per CPU, or core count, varies widely across market segments and vendors. There is no magic number that fits all necessities. At the time of writing, CPUs in the HPC market house from 16 up to 64 cores.

The CPU package also hosts the first levels of the memory hierarchy (up to the L3 cache) and a network that connects all the components inside the package. The CPU is inserted into a socket, which connects it to a bigger system. The socket bridges the connection between the CPU and other components such as the main memory, accelerators, and other sockets. In HPC it is common to use CPU and socket interchangeably. For example, a CPU package may have 32 cores but we can also say that there are 32 cores per socket.

We call node the computational element sold by hardware providers and integrators. A node is the basic unit of an HPC system. The node contains one or more sockets and slots to connect modules such as memory and network cards. The connection grid and placement of components in a node board can affect the overall performance of the system. The memory modules are commonly connected to a single socket of the node. In this case, all CPUs can access all memory modules, but the access time to modules that are connected to a different socket is higher than the access time to a memory module that is connected to the socket directly. We call these memory scheme Non Uniform Memory Access (NUMA) and may impact the performance of the execution.

Figure 3.3 shows a picture of a compute node in MareNostrum4, the flagship supercomputer installed in BSC. The image shows a node with two sockets, each one with eight memory modules. The node also has a 240GB Solid State Drive (SSD) local storage.

Scaling out

We say that an HPC system is scaling out when interconnects more than one node. The connection between nodes is usually done through a network. The rate at which data is transferred between nodes is called network bandwidth and it is measured in B/s. There are different commercial network technologies that are used in HPC systems. Each network technology has a given peak network bandwidth, which should be disclosed by the vendor. The network topology has also an impact on the overall performance of the system. Depending on the network topology, the system is more or less resilient to system failures and network congestion. Please note that HPC systems are usually shared between multiple users, which makes it difficult to predict the network traffic at any given time.

Multiple nodes can be grouped together in a single physical structure called rack. Apart from compute nodes, a rack may also have one or more network switches, management modules, Power Supply Units (PSUs) and other features. Figure 3.4 shows a picture of a compute rack of MareNostrum4.

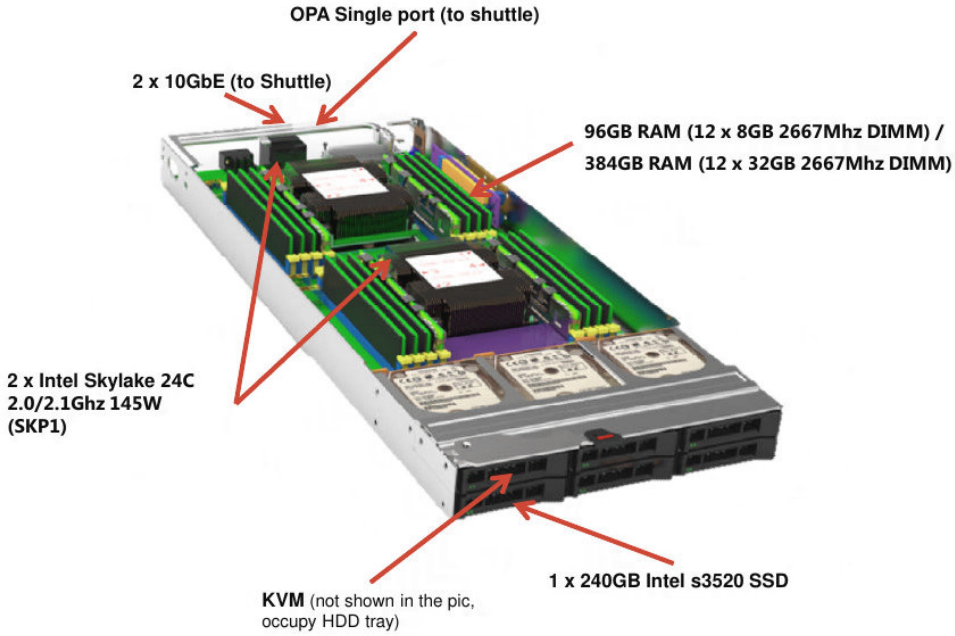

Figure 3.3: Annotated picture of a compute node in MareNostrum4

Figure 3.4: Picture of a MareNostrum4 compute rack

3.2 Software stack

Compilers

The compiler is in charge of translating high level human-readable code into machine code, or binary, which is a representation of the sequence of instructions mentioned in Section 3.1. The translation process is generally called compilation and has multiple steps. For each step, there is a specific utility. Hence, we should refer to the set of utilities as toolchain, although the terms compiler and toolchain are commonly used interchangeably. In this thesis, I only study the final product of a compilation, comparing the resulting binary with the source code.

We divide compilers in two categories: open source and vendor specific compilers. Open source compilers are generally available to the public and cover a wide range of CPU architectures. The most notable open source compilers are the GNU Compiler Toolchain and LLVM. On the other hand, vendor specific compilers are provided by the CPU vendor and they usually require a license to be used. The selling point of vendor compilers is that they focus on a specific CPU architecture, which should mean that are able to better leverage the computational resources of the micro-architecture of that CPU.

During the compilation of a program there are one or more optimization phases. These phases apply different techniques to produce a binary which yields a better performance. The user can, at some degree, specify which optimizations should be performed by passing to the compiler a set of optimization flags. Each compiler has a wide range of optimization flags, which makes it difficult to the end user to select the ones that are of most interest. To make matters worse, two compilers may have a flag with the same name that trigger completely different optimization options. The most common practice is to use a set of compiler flags given by the system provider or the system administrator. In Chapter 5, I study the effect of the recommended optimization flags.

Arithmetic libraries

Arithmetic libraries implement widely used arithmetic routines such as matrix operations, signal processing functions, etc. As with compilers, arithmetic libraries can be open source or vendor specific. In this thesis I present a performance study of Alya, an application which does not require external dependencies to arithmetic libraries. For this reason, I do not cover any arithmetic library.

Programming models

Programming models are the paradigm that programmers use to leverage the resources in a parallel system. They govern how parallel applications are written. There are multiple programming models in HPC but we can classify them in two major categories.

Shared memory programming models are based in threading. A process has multiple threads that execute in parallel. The execution within a thread is sequential but there is no implicit synchronization across threads. In this programming model, all threads share a virtual memory space, meaning that they can access the same memory locations without the need for explicit data exchange between threads. pthreads is a UNIX library that implements a shared memory programming model. The programmer manages parallel regions by spawning and explicitly synchronizing threads via calls to the pthread library. OpenMP is a shared memory programming model based on code annotation. The programmer adds pragma directives that do not interfere with the sequential code. Please note that OpenMP is a standard and has multiple implementations. It is common for compiler toolchains to include an OpenMP implementation in the form of a runtime library. For example, the GNU compiler toolchain ships with the `libgomp` library and the Arm HPC Compiler toolchain ships with the `libomp` library. Both libraries are implementations of the same standard. At compile time, if the appropriate OpenMP flag is used, the compiler identifies the OpenMP directives and injects the necessary calls to the underlying threading library. A priori, a code with OpenMP directives can both be compiled with and without support for OpenMP and execute just fine.

Distributed memory programming models are based on processes that have isolated virtual memory spaces. This means that a process cannot access data from another process without some kind of explicit data exchange primitive. Message Passing Interface (MPI) is a distributed memory programming standard. It is based on point-to-point ant collective operations that programmers invoke in their code. Point-to-point communications exchange data between a pair of processes while collective communications can involve more than two processes. Similar to OpenMP, MPI is a standard with multiple implementations. The most common libraries that implement the MPI standard are OpenMPI, mpich, and Intel MPI.

The programmer might also combine both shared and distributed memory programming models. We call this a hybrid programming model. A common example of this in HPC is a program with both OpenMP and MPI parallelization. In this thesis, I use a performance model for MPI-only applications. For this reason, I only cover this specific programming model.

As a concluding remark, I would like to point out that there are other programming models specifically designed for heterogeneous systems with different hardware accelerators. In this study, I do not focus on any hardware accelerator so these programming models are outside of the scope of the thesis.

Performance counters

Performance counters are hardware elements that measure events that occur inside the CPU, such as number of instructions executed, number of accesses to a given cache level, etc. These counters are the closest a programmer can get to measure a bare metal-system. The set of performance counters available on a CPU vary greatly depending on the micro-architecture, the architecture, the system integrator and the system administrator. Two CPUs with the same model may have a different set of performance counters available due to how the system administrator has configured the system. Furthermore, the method and the latency to read these performance counters can be drastically different depending on the CPU architecture. Most ISAs have specific instructions to access performance counters.

perf is a UNIX utility to access performance counters and it is portable across architectures. This tool defines a list of generic performance counters that match with the real hardware performance counters. Thanks to this layer of abstraction, the programmer can access with the same alias a wide set of performance counters that perf maps to the correct hardware counters depending on the CPU architecture.

Performance Application Programming Interface (PAPI) is a library that leverages the portability of perf and has an easy programming interface. PAPI also defines a list of generic counters called *presets* that should be available on most CPUs. However, and important comment is that, even with the same name, PAPI counters may not measure the same event when read in different systems. For example, `PAPI_VEC_INS` may read all issued vector instructions on a CPU while only measuring issued arithmetic vector instructions on another system. Furthermore, there have been cases were the hardware implementation of the performance counter, or the PAPI library were not implemented correctly. The authors in [34] show that the `PAPI_VEC_INS` counter in a ThunderX CPU was not matching the specifications of the Armv8 ISA.

All in all, PAPI is the de-facto standard for reading performance counters in production systems. In this thesis, I perform some measurements using the PAPI library and present them in Chapter 5. On top of that, I present a performance study of a scientific application using performance monitoring tools that leverage the underlying PAPI library.

3.3 Machines under evaluation

Dibona - Arm

Dibona is the final prototype developed during the Mont-Blanc3 European Project. The cluster was integrated by ATOS/Bull with their Sequana infrastructure. At the time of deployment, the system has 40 compute nodes. A preliminar study of the system was done in [2]. The work was expanded upon and published in [3]. The compute nodes of the Dibona cluster house two Marvell ThunderX2 CPUs. Figure 3.5 shows a schematic view of a compute node in Dibona. All data related to Dibona in this thesis is color-coded in red and labeled as *mb3*. Figure 3.5 shows a schematic view of a node in Dibona.

MareNostrum4 - Intel

MareNostrum4 is the flagship Tier-0 supercomputer hosted at BSC. It has 3456 nodes. The authors in [2] use MareNostrum4 as a baseline to compare with Dibona. The cluster ranked 29th in the Top500 of June 2019. The compute of the MareNostrum4 cluster house two Intel Xeon Platinum 8160 CPUs. Figure 3.6 shows a schematic view of a compute node in MareNostrum4. All data related to MareNostrum4 in this thesis is color-coded in blue and labeled as *mn4*. Figure 3.6 shows a schematic view of a node in MareNostrum4.

Power9 - IBM

Power9 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 [35], 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*. Figure 3.7 shows a schematic view of a node in Power9.

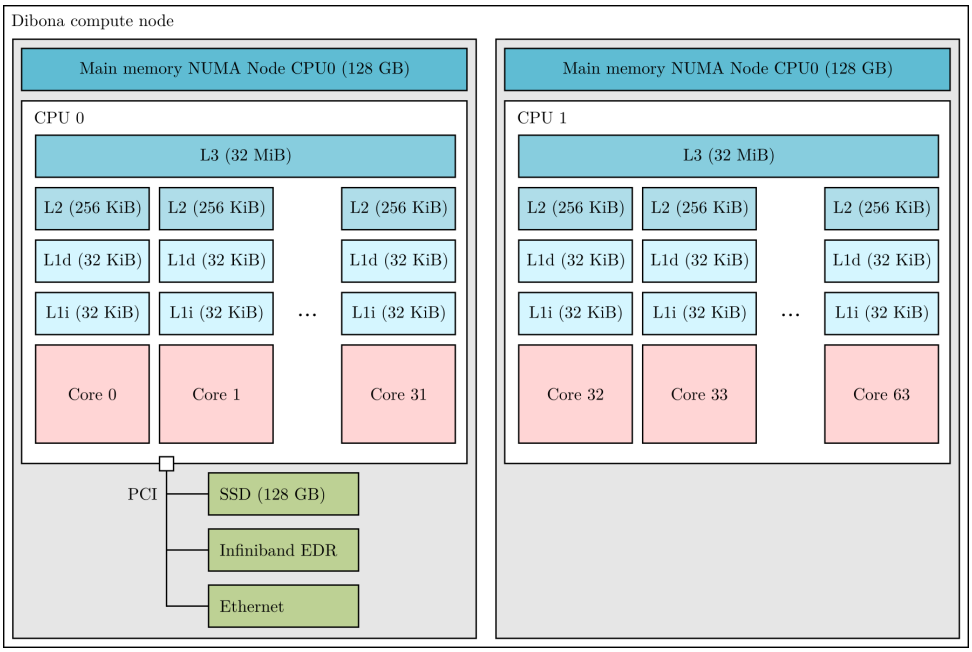

Figure 3.5: Schematic view of a node in Dibona

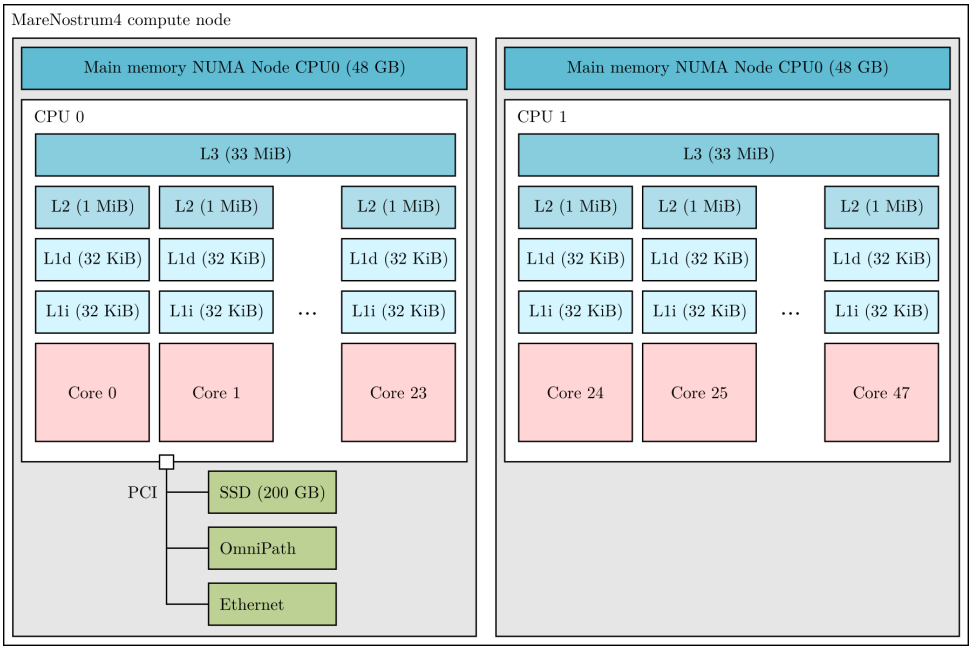

Figure 3.6: Schematic view of a node in MareNostrum4

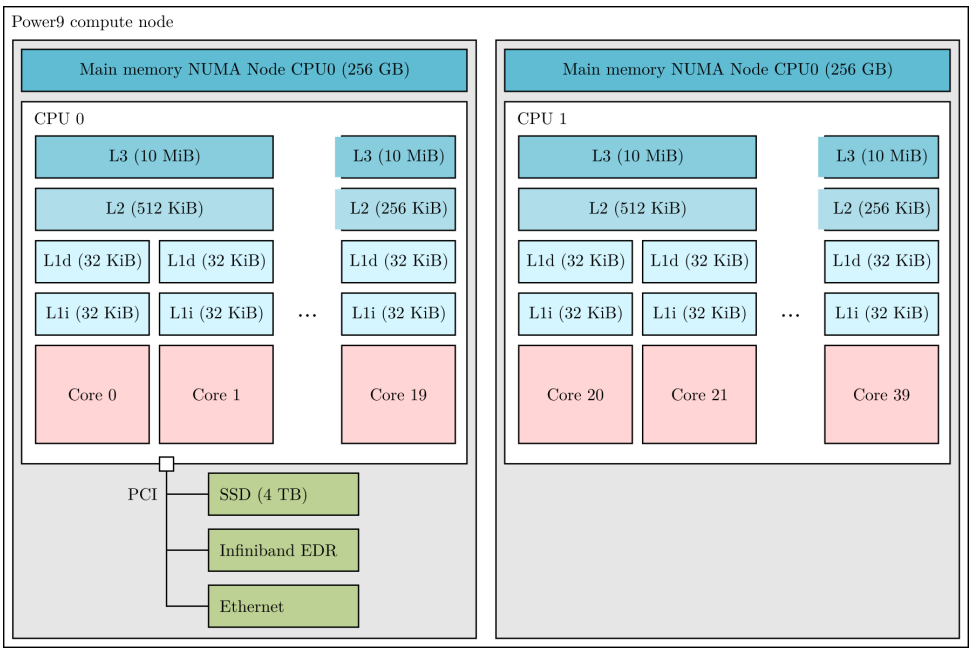

Figure 3.7: Schematic view of a node in Power9

Hardware and software comparison

Table 3.1 lists all the relevant hardware features and system software tools across the three machines I study in this thesis.

3.4 Roofline model

Floating-point throughput

Let I be the number of instructions issued per cycle; O the number of floating point operations per instruction; and f the frequency of the processor. The theoretical peak throughput of the FPU, F_p can be computed as shown in Equation 3.1.

$$F_p = I \times O \times f \tag{3.1}$$

Modern processors have SIMD units which can process more than one floating point element per instruction. We can define the number of elements per SIMD registers, S[1], and extend the previous equation to account for SIMD registers. Equation 3.2 models the theoretical peak performance of a system with SIMD instructions.

$$F_p = I \times O \times f \times S \tag{3.2}$$

The methodology I present in this thesis, tries to measure the peak floating point of a system empirically. HPL is, historically, the most common benchmark to report floating-point performance in HPC. The reference code has been parallelized with MPI and OpenMP. The performance reported by HPL approaches the theoretical peak of the machine. In the Top500 list of November 2019, Summit reaches 75% of its theoretical peak. Even if HPL is a very compute intensive benchmark, its performance does not depend exclusively on the floating-point throughput. There are other factors such as parallelization schemes and network bandwidth that may have a great impact on performance.

In this thesis, I present a custom micro-benchmark which depends exclusively on the floating-point performance of the system. It also does not have dependencies with external math libraries. The micro-benchmark is called FPU_μKernel and measures performance by executing a sequence of fused-multiply-accumulate assembly instruction with no data dependencies between them. Section 4.1 details the implementation of the FPU_μKernel and the results obtained in three systems with different architectures.

[1] On some architectures (e.g., RISC-V), S is also called *vlen*

	Dibona	MareNostrum4	Power9
CPU name	Marvell	Skylake	IBM Power9
	ThunderX2	Platinum	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
System integrator	ATOS/Bull	Lenovo	IBM
GNU Compiler	8.2.0	8.1.0	8.1.0
Vendor Compiler	Arm 19.0	Intel 2017.4	XL 16.1.1.2
Other Compiler			PGI 18.10
MPI (OpenMPI)	3.1.2	3.1.1	3.1.1
PAPI	5.5.1.0	5.7.0	5.6.0
Extrae	3.5.4	3.7.1	3.7.1

Table 3.1: Hardware and software configuration of the HPC platforms

Memory bandwidth

Let M be the number of Bytes transferred to/from a given memorization element (i.e., caches or main memory) and t the time it takes to complete the data transfer. We define the memory bandwidth, B, as $B = M/t$. In contrast to the theoretical peak throughput of the FPU, there is no simple answer to what is the theoretical peak memory bandwidth of a memorization element. For the main memory, the vendor should provide a transfer rate which depends on the memory technology that the memory module uses. This transfer rate, B_0 is usually expressed in Transfers per second (T/s), which can be converted into B/s. Given that a CPU has N memory modules connected via memory channels, we can model the theoretical peak memory bandwidth, B_p of the main memory as shown in Equation 3.3. Please note that Equation 3.3 assumes that all memory nodes are of the same technology.

$$B_p = B_0 \times N \tag{3.3}$$

On multi-socket systems, a CPU can also access the memory modules that are connected to remote sockets. As mentioned in Section 3.1, this remote accesses are slower compared to the accesses to the local NUMA node. For simplicity, I do not consider this effect when calculating the peak memory bandwidth.

For the cache levels, there is almost no information regarding its theoretical peak bandwidth. The only way to know B_p is to ask the vendor directly. In some cases, we can design experiments to infer the peak memory bandwidth empirically. In Sections 4.2 and 4.3, I present two state-of-the-art micro-benchmarks that aim to measure the peak memory bandwidth across the whole memory hierarchy.

Arithmetic intensity

The arithmetic intensity I is defined as the number of floating point operations executed per each Byte of data transferred to/from memory. Williams et al. in [36] point out the difference between *Operational intensity* and *Arithmetic intensity*. The term *Arithmetic intensity* takes into account the data transferred from CPU and cache, whereas *Operational intensity* also includes the traffic between caches and the main memory. The authors also mention that *Operational* is a broader term which may include non-arithmetic workloads. For the rest of the document I use the term arithmetic intensity. The methodology proposed in this thesis takes into account data transferred between CPU, caches and main memory.

Basic roofline model

The roofline model characterizes the floating point performance of a machine in function of the arithmetic intensity of a code. Given a peak floating point performance F_p and a peak memory bandwidth B_p of a system, the attainable performance $F_s(I)$ is defined as follows:

$$F_s(I) = min(F_p, B_p \times I) \tag{3.4}$$

This function can be represented as a curve in a two-dimensional plane where the x-axis is the arithmetic intensity and the y-axis is the floating point performance. The curve has two very distinct regions. When $F_s(I) = B_p \times I$, the plotted function is a straight line with positive slope. In this region, the attainable performance is limited by the peak memory bandwidth of the system. When $F_s(I) = F_p$, the plotted function is a straight line with zero slope. In this region, the attainable performance is limited by the peak floating point performance. The point where $B_p \times I = F_p$ is called *machine balance* and we denote it as I_r. We can reformulate the roofline model in function of I_r:

$$F_s(I) = \begin{cases} F_p & \text{if } I > I_r \\ B_p \times I & \text{if } I \leq I_r \end{cases} \tag{3.5}$$

We can measure the arithmetic intensity of a given problem, I_0, and place it along the x-axis of the roofline model plot. The value $F_s(I_0)$ is the attainable performance of that given problem in the machine characterized by the roofline model. If $I_0 < I_r$, we say that the problem is memory bound. On the other hand, if $I_0 > I_r$, we say that the problem is not memory bound. When a problem is memory bound, it means that more floating point resources do not provide any benefit to the attainable performance.

Extensions to the roofline model

Ilic et al. in [37] present an extension of the roofline model. Their work includes the bandwidth of the multiple levels of caches. Each one of the cache levels adds to the plot a new straight line with positive slope. This extended model gives even more insight on how the memory hierarchy can improve performance in memory bound workloads. Marques et al. in [38] apply the cache-aware roofline model to an Intel system using the Intel Advisor. Their work show how the model helps detecting performance bottlenecks and optimization hints. Unfortunately, the Intel Advisor is an architecture specific tool. We employed a more portable and generic tool to characterize machines implementing different ISAs with the roofline model.

3.5 Efficiency model

In this thesis I leverage the metrics for modeling the performance of parallel applications developed and promoted by the European Center of Excellence (CoE) POP [39]. 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 Chapter 6 I focus on analyzing the parallel efficiency of an MPI application. 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.

The efficiency model follows the same steps to analyze every 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 Analysis (FOA). This 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. 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. Note that for the multi-node study we used always full nodes. Guided by the results on the metrics determine the issues that limit the scalability of the code on multiple nodes.

The study I present in this thesis is done with strong scalability, but the methodology and metrics can be applied also to weak scaling codes.

Metrics for Performance Analysis

For the definition of the POP metrics needed in the rest of the thesis we use the simplified model of execution depicted in Figure 3.8.

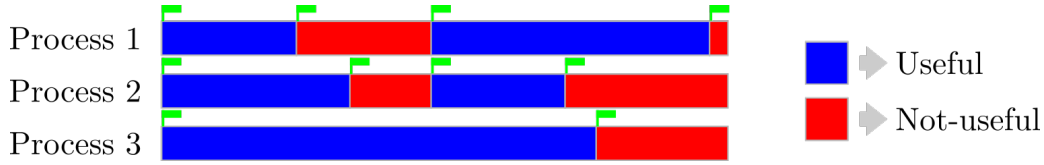

Figure 3.8: Example of parallel execution of 3 MPI processes.

We call $P = \{p_1, \ldots, p_n\}$ the set of MPI processes. 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 computation (the set of the blue intervals). We define the sum of the durations of all useful time intervals in a process p as:

$$D_{U_p} = \sum_{U_p} \blacksquare = \sum_{j=1}^{|U_p|} u_j^p \tag{3.6}$$

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:

$$L_n = \frac{\text{avg}_{|P|}\left(\sum_{U_p} \blacksquare\right)}{\max_{|P|}\left(\sum_{U_p} \blacksquare\right)} = \frac{\sum_{i=1}^{n} D_{U_i}}{n \cdot \max_{i=1}^{n} D_{U_i}} \tag{3.7}$$

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:

$$T_n = \frac{\max_{|P|} t'_p}{\max_{|P|} t_p} \tag{3.8}$$

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:

$$S_n = \frac{\max_{|P|} D_{U_p}}{\max_{|P|} t'_p} \tag{3.9}$$

Where D_{U_p} is computed as in Eq. 3.6 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 3.8, 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:

$$U_n = \frac{\sum_{P_0} D_{U_p}}{\sum_{P} D_{U_p}} \tag{3.10}$$

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 Cycle (IPC)).

Finally we can combine the efficiency metrics introduced so far in the *Global Efficiency* defined as $G_n = P_n \cdot U_n$. Figure 3.9 shows the hierarchical structure of the efficiency model.

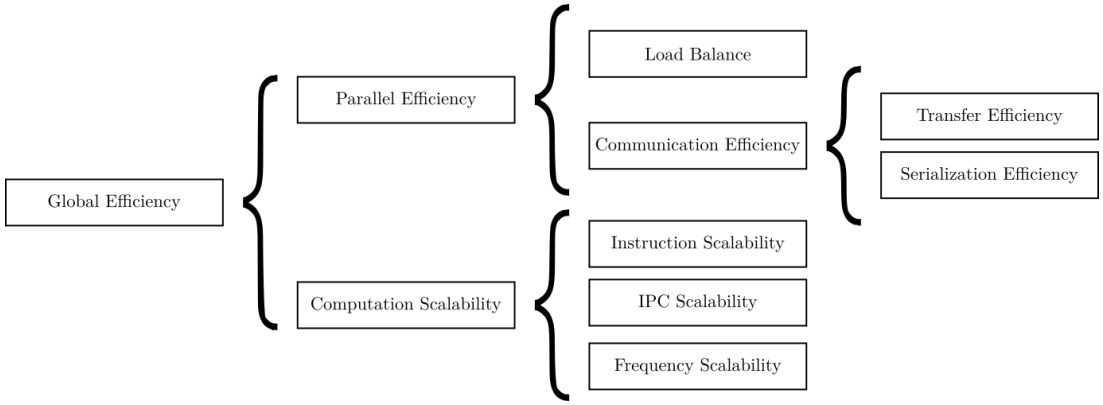

Figure 3.9: Hierarchical structure of the POP performance model

3.6 Performance analysis tools

In this thesis, I use the same methodology as in [4] to measure the POP metrics. I use BSC's performance evaluation tools Extrae, Paraver, and Dimemas. Measuring the efficiency metrics empirically might produce metrics above the ideal one or 100%. We treat this results as if they were equal to the ideal value.

Extrae

Extrae is a tracing tool which intercepts MPI calls of an application and collects runtime information. The information gathered includes performance counters, which MPI primitive was called and which processes were involved during the communication. All this information is stored into a file called a trace. Tracing MPI applications with Extrae does not require to recompile the application. Since Extrae performs its measurements during runtime, it is bound to overheads. The amount of overhead depends on multiple factors, mainly: number of communicating processes, number and frequency of MPI calls, amount of data collected at each interception. We are still developing a method to quantify the tracing overhead. At the moment of writing, a general rule of thumb is to discard all measurements derived from bursts shorter than ten micro-seconds.

Paraver

Paraver [40] is a visualization tool that helps navigate the traces generated by Extrae. A common visualization mode when using Paraver is the timeline. Timelines represent the evolution of a given metric across time. Figure 3.10 shows two examples of timelines. The x-axis represents time and the y-axis represents the MPI processes. Each burst is color coded. For quantitative values (left timeline), the color scale goes from dark blue (high values) to light green (low values). For qualitative values (right timeline), each color represents a different concept (e.g., MPI primitive). We sometimes omit certain regions of the execution from the timeline. For example, the top timeline in Figure 3.10 shows the number of instructions per burst. I only included the bursts that perform useful computation and omitted all bursts which correspond to MPI calls. Omitted burst are shown in white.

Dimemas

Dimemas [41] is a simulation framework for Paraver traces. It takes a trace as an input and generates a new trace where all communications have an ideal latency and bandwidth. For blocking communications, this means that the communication bursts will end when the slowest process arrives to the communication. For non-blocking communication, the bursts shrink to as if they had a duration of zero. Please note that Dimemas does not require to repeat the execution. It only manipulates an already existing Paraver trace file.

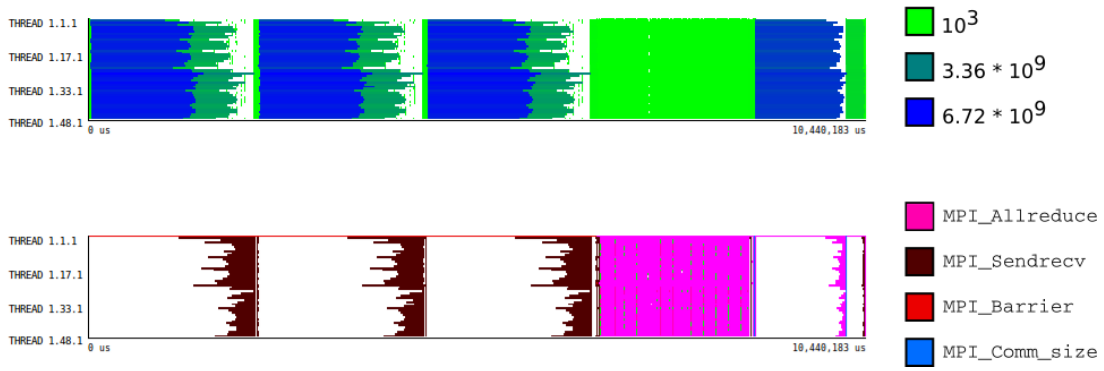

Figure 3.10: Paraver timeline examples. Top: quantitative representation of number of instructions. Bottom: MPI primitive calls.

Clustering

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.

Representation of the efficiency model

The methodology I present in this thesis uses the same representation method presented in [4]. We measure the POP metrics in traces with increasing number of MPI processes and we present them in a table where each row represents a different metric and each column represents a run with different number of hardware recourse (in this study, MPI processes). Table 3.2 shows an example of these tables. Each cell contains the value of a given metric and is color coded. We consider a good efficiency value to be between 100% and 90%, a middling efficiency between 90% and 80%, and a bad efficiency below 80%. We do not gain insight by observing a single value of a given metric, but rather by studying the trend of the metric. Efficiency metrics are studied at two scales: single node and multiple node analysis. The single node analysis focuses on how contention of resources within a node can affect the performance of each process. This effect is typically observed with a degrading IPC scalability. When scaling within a multi-socket node, it is important to acknowledge how processes are distributed across sockets. Unless stated the contrary, we always distributed the processes evenly between sockets. The multiple node analysis focuses on how well the application can scale beyond a single node. The parallelization scheme and/or the communication network are the typical performance bottlenecks in this case. The effects of resource contention are sometimes hidden in the multiple node analysis. This is because the metrics labeled with *Scalability* are relative to a baseline run. In the case of multiple node analysis, the base line is usually a run with two whole nodes. At the baseline point, the resources within a node are already saturated.

	2	4	8	12	24	48
Global efficiency	83.20	82.95	81.67	81.74	79.79	75.13
├ Parallel efficiency	83.20	83.17	82.31	83.01	83.01	84.12
│ ├ Load balance	83.59	83.63	82.71	83.35	83.33	84.75
│ └ Communication eff.	99.53	99.45	99.52	99.59	99.62	99.25
│ ├ Serialization eff.	99.93	99.85	99.89	99.94	99.96	99.85
│ └ Transfer eff.	99.61	99.60	99.62	99.65	99.67	99.41
└ Computation scalability	100.00	99.74	99.23	98.47	96.12	89.32
├ IPC scalability	100.00	99.74	99.25	98.47	96.15	89.24
├ Instruction scalability	100.00	100.00	100.00	100.00	100.00	100.10
└ Frequency scalability	100.00	100.00	99.98	100.00	99.97	99.99
Speedup	1.00	1.00	0.98	0.98	0.96	0.90
Average IPC	2.14	2.14	2.13	2.11	2.06	1.91
Average frequency [GHz]	2.09	2.09	2.09	2.09	2.09	2.09

100%
95%
90%
85%
<80%

Table 3.2: Example of how we represent the efficiency metrics on a table

As a last note, I would like to address how to apply the single node analysis to applications which do not fit a single node due to memory constraints or require more MPI processes than cores in a node. A possible solution would be to fix the number of MPI processes to the minimum required, N. Consider a system with nodes with C cores. The baseline run would require $N/2$ nodes, having two process per node. Successive runs would double the number of processes per node until filling a whole node. The i-th case would require $N/2i$ nodes with $2i$ processes per node. Please note that this solution has limitations. If N is too big, the baseline run with two processes per node would be a waste of HPC resources. For this reason, I suggest choosing a baseline with a higher number of processes per node.

Chapter 4

Micro-benchmarks

Entia non sunt multiplicanda praeter necessitatem.

"More things should not be used than are necessary."

— William of Ockham

The first layer of abstraction of the methodology I present evaluates the system as close to a bare-metal environment as possible. This layer is comprised of a set of codes which stress a specific component of the system. The goal is to have a set of simple tools that can measure the quantities that characterize the architecture of HPC clusters.

In Section 4.1, 4.2 and 4.3 I present the measurements of the quantities (i.e., peak floating point performance and memory bandwidth) needed to derive the roofline model of each machine. The roofline itself is presented in Section 4.4. This model gives insight on the empirical boundaries under which a scientific applications operates (Chapter 6 elaborates on this). In Section 4.5 I present a method to measure latency across the memory hierarchy. Finally, I focus on the performance of the communication network. I model the communication network as different software layers that build upon each other. Section 4.6 studies the first software layer with the tools provided by the manufacturer (i.e., `ibverbs`). There are other software layers that can influence performance in communication bound applications. The second software layer corresponds to the MPI implementation, which I discuss in Chapter 5.

4.1 FPU and SIMD performance

We designed a micro-kernel to measure the peak floating point throughput of the machine. We call this code FPU_μKernel and contains exclusively fused-multiply-accumulate assembly instructions with no data dependencies between them. The kernel has four versions distinguishing between *i)* scalar and vector instructions; and *ii)* single and double precision.

We started with a simple C statement `a * b + c` and observed which instructions were generated by the compiler. Table 4.1 shows the assembly instructions used in the micro-kernel. Although the x86 ISA has floating point instructions that run on the FPU, it is recommended to use the more recent SIMD instructions so the compiler uses `VFMADD132SS` and `VFMADD132SD` for single and double precision, respectively. This means that the scalar version of the code in the x86 architecture uses vector instructions with the same behavior as scalar floating point instructions.

Figure 4.1 shows results of the FPU_μKernel on one core of each machine. We observe a higher scalar floating point performance in Power9 than in Dibona and MareNostrum4. This is because the CPUs in Power9 run at a higher frequency and have a higher instruction throughput. For the vector versoin of the micro-kernel, 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. This results on $\times 4$ and $\times 2$ performance improvement for single and double precision, respectively, compared to Dibona. The FPU_μKernel shows us that the floating point performance is closely related to the clock frequency of the CPU and the width of the SIMD registers.

Figure 4.1 only accounts for the single-core performance of each machine. The theoretical peak floating point throughput of a whole node can be computed as the single-core performance times the number of cores in a node. For example, the FPU in one Marvell ThunderX2 core reaches 8 GFlop/s for double precision arithmetic. Each node in Dibona houses two TX2 CPUs with 32 cores each, which amounts to a total of 512 GFlop/s.

Machine	Instruction	Type
Dibona	`FMADD Sx,Sx,Sx,Sx`	FPU SP
MareNostrum4	`VFMADD132SS mmx,mmx,mmx`	FPU SP
Power9	`FMADDS x,x,x,x`	FPU SP
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.4S,Vx.4S,Vx.4S`	SIMD SP
MareNostrum4	`VFMADD132PS mmz,mmz,mmz`	SIMD SP
Power9	`XVMADDASP x, x, x`	SIMD SP
Dibona	`FMLA Vx.2D,Vx.2D,Vx.2D`	SIMD DP
MareNostrum4	`VFMADD132PD mmz,mmz,mmz`	SIMD DP
Power9	`XVMADDADP x, x, x`	SIMD DP

Table 4.1: Instructions in the FPU_μKernel

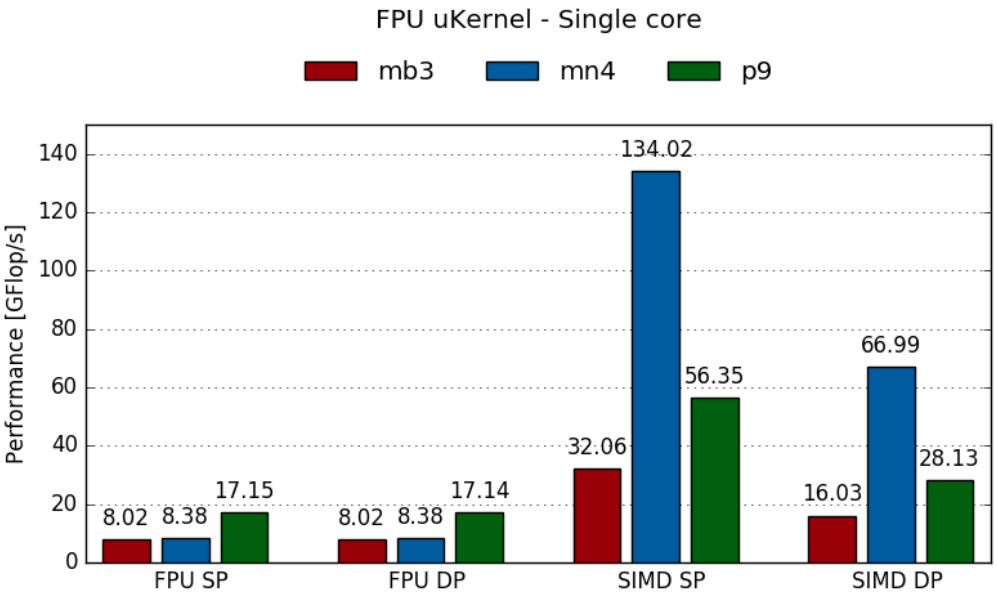

Figure 4.1: Sustained performance in one core of the four versions of the FPU_μKernel

4.2 STREAM

STREAM is a simple synthetic benchmark to measure sustainable memory bandwidth. The program is structured in four distinct computational kernels: Copy, Scale, Add and Triad. The reference version of the benchmark includes a parallel version using OpenMP. We used the STREAM benchmark (v5.10.1) leveraging an OpenMP parallelization.

The kernels iterate through data arrays of double precision floating point elements (8 B) with a size fixed at compile time. It is required that the size of each array must be at least four times the size of the sum of all the last-level caches or ten million elements (the maximum value between the two).

$$E \geq max\ \{4 \cdot S/8\ ,\ 10000000\}$$

Where E is the number of elements of each array (refered to as `STREAM_ARRAY_SIZE` in the code); and S is the size of the last level caches in Bytes. Table 4.2 shows the minimum value for E on one node of Dibona, MareNostrum4 and Power9. These are the values used to perform our tests.

Table 4.2: E minimum values for each cluster

Machine	S	E
Dibona	67108864	33554432
MareNostrum4	69206016	34603008
Power9	209715200	104857600

We run the benchmark by fixing the problem size to the minimum valid value of E for each platform as reported in Table 4.2 and increasing gradually the number of OpenMP threads. Threads are pinned to cores by using `OMP_PROC_BIND=true` and distributed evenly between both sockets of the node.

Figure 4.2 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.

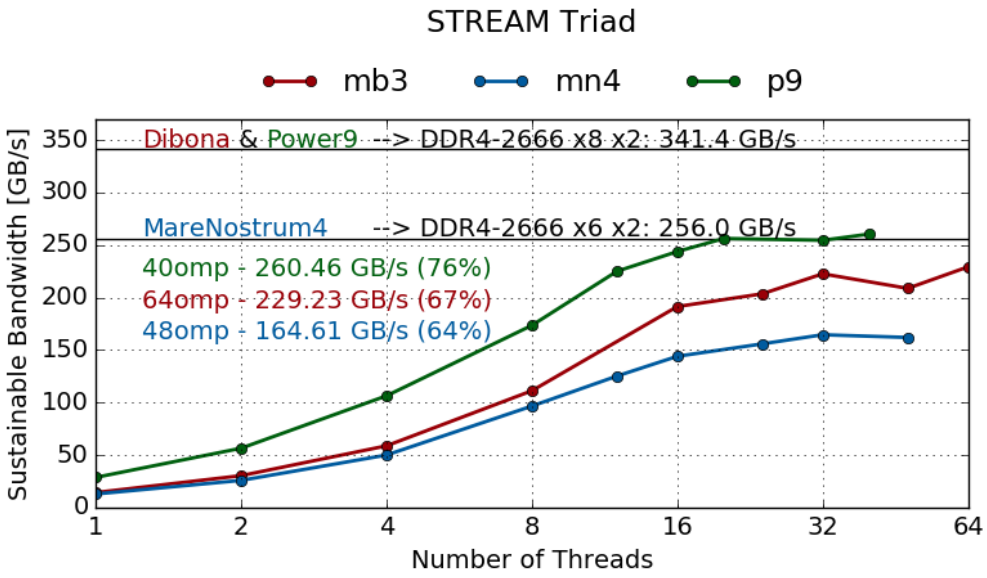

Figure 4.2: STREAM Triad on one node of Dibona, MareNostrum4 and Power9

We observe that all three systems have a similar behavior when running STREAM. The achieved bandwidth increases when adding more threads. This is because the memory petitions from each thread can be served by the different memory controllers in the CPU. There is a point in which adding more threads fills the capacity of the memory controllers of serving requests to the memory making the memory bandwidth reaching a plateau. In this situation, we say that we saturate the memory bandwidth.

Dibona reaches around $\sim$ 230 GB/s (67% of the peak) in the Triad kernel when running with 64 threads. MareNostrum4 gets $\sim$ 165 GB/s (64% of the peak) with 48 threads. Power9 reaches $\sim$ 260 GB/s (76% of the peak) with 40 threads. Power9 has a significantly different memory hierarchy than the other two systems. It also runs at a higher frequency. Both of these factors may be the cause of it reaching a higher bandwidth with respect to the theoretical peak.

4.3 lmbench

lmbench is a collection of benchmarks designed by Larry McVoy and Carl Staelin [42]. A full run of all the benchmark suite analyzes the machine's hardware capabilities: integer and floating point performance, memory latency and bandwidth, disk access performance, network latency, and OS noise.

In this section, we report the results obtained with the "Memory Read Bandwidth" test within lmbench. This test measures the memory hierarchy's load bandwidth. In contrast with STREAM, lmbench results clearly show memory bandwidth measurements at different cache levels. The author of STREAM published in 2003 a followup version, STREAM2. This version is designed to measure sustained bandwidth at all levels of the memory hierarchy. Like STREAM, STREAM2 has multiple kernels. Each kernel yields a different memory bandwidth since it has a unique memory access pattern. Since the multiple kernels of STREAM2 do not provide a unified result, I decided to use lmbench to measure the memory bandwidth across the memory hierarchy.

The memory load bandwidth test takes a size S, in MiB, a number of processes P, and a number of repetitions N as input parameters. The output of the test is the sustained bandwidth of P threads performing sequential reads to an array of size S averaged across N traversals of the whole array. lmbench manages threading with POSIX threads.

If we perform successive runs with constant P and N, but increasing S, we obtain the sustained read bandwidth of the different levels of the cache hierarchy. When S is greater than a given cache size, the program will be bottlenecked by the next level in the memory hierarchy.

Figures 4.3, 4.4, and 4.5 show the sustained memory bandwidth measured in Dibona, MareNostrum4, and Power9 respectively. The x-axis represents the problem size S and the y-axis represents the memory bandwidth as reported by the test. Each point in the figures represents one run of the test with $N = 10$. Each line represents a different sequence of runs with a given number of processors P. For simplicity, I only included the runs with one processor, half a node, and a full node. Threads are always spread evenly across sockets. The vertical lines indicate a jump up in the memory hierarchy. These lines divide the plot into four regions. From left to right: L1 cache, L2 cache, L3 cache, and main memory. As confirmation, the highest memory bandwidth in the main memory region is the same as the best bandwidth measured with STREAM in Section 4.2.

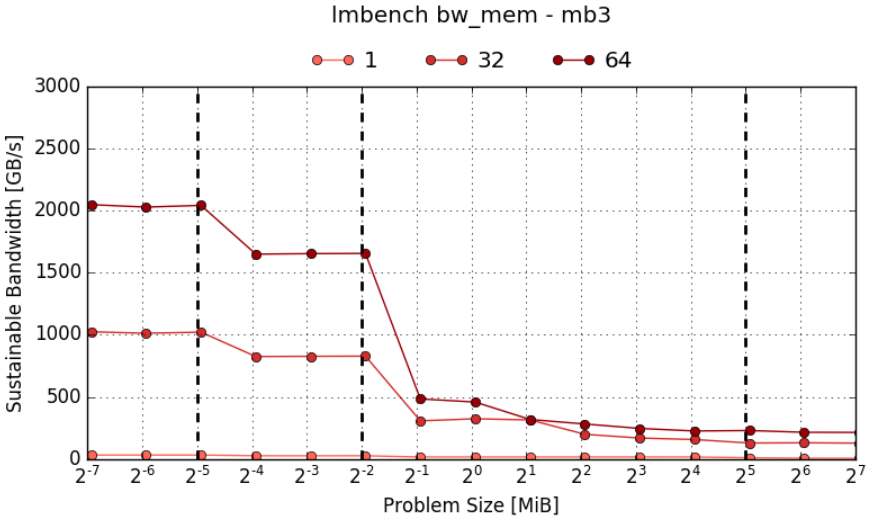

Figure 4.3: lmbench Memory load Bandwidth in Dibona

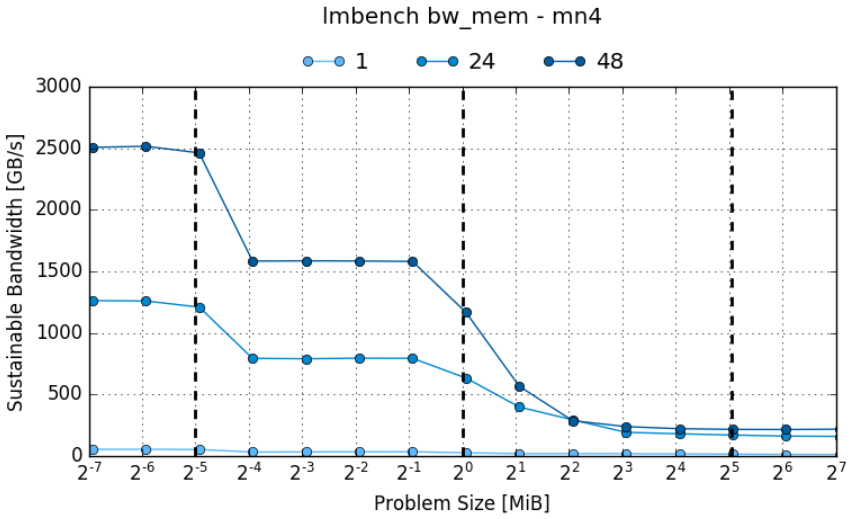

Figure 4.4: lmbench Memory load Bandwidth in MareNostrum4

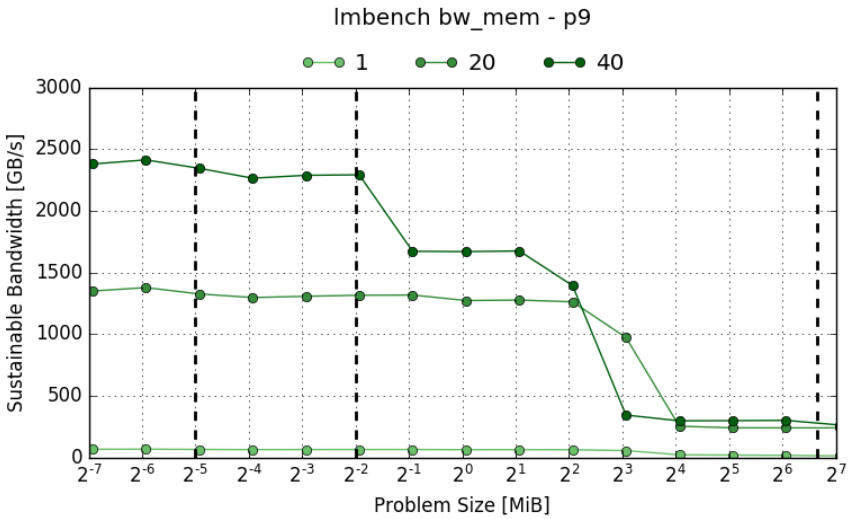

Figure 4.5: lmbench Memory load Bandwidth in Power9

First of all, we observe a drop in bandwidth each time that the problem size crosses one of the vertical lines. The drop is not always exact. For Dibona, it clearly occurs when the problem size S crosses the boundaries of the L1 size and the L2 size. In MareNostrum4, it also occurs from L1 to L2, but it has a smoother transition when going from L2 to L3. I do not have an exact explanation of this behavior but it might be due to the cache allocation and replace policies. Power9 has a very different behavior than the other systems. When running with the whole node, there is no drop when going from L1 to L2 caches. Moreover, with half of the node, there is no drop from L2 to L3 cache. This is most likely due to the cache hierarchy in the Power9, which has L2 and L3 caches shared across pairs of cores. The bandwidth drops substantially with $S > 4$ MiB. This is because, even if the total size of the L3 cache is 100 MiB, the amount of L3 cache per pair of cores is 10 MiB, which makes tests with a bigger problem size go to main memory.

We also observe that Dibona has a lower L1 cache bandwidth, $\sim$ 2000 GB/s, compared to MareNostrum4 and Power9, both at $\sim$ 2500 GB/s. This means that programs which are memory bound, but have a good L1 cache locality, might perform worse in Dibona. The L2 cache bandwdidth in Dibona and MareNostrum4 drops to $\sim$ 1600 GB/s. Similar to our observation for the L1 cache, the drop in L2 cache bandwidth might show a better performance in Power9 for programs which exploit L2 cache locality.

In conclusion, we have a tool to measure memory bandwidth at different cache levels. Since vendors do not provide nominal bandwidth for each level in the memory hierarchy, I use the results of the lmbench test to construct the roofline model.

4.4 Roofline model

We could construct the roofline model of a machine following its technical specification. The floating point throughput of a CPU and the memory bandwidth of a DRAM chip are widely advertised by vendors. However, vendor specifications may not reflect what the final user can achieve. For this reason, it would be useful to construct the roofline model from empirical measurements using micro-benchmarks.

Yu et al. present in [43] their Roofline Toolkit. A tool to construct the empirical roofline model of a machine. Their framework is flexible and can be extended to multiple CPU architectures. It also supports systems with GPUs. The Roofline Toolkit consists on running a simple kernel written in C which has a compile-time parameter that controls its arithmetic intensity. The tool constructs the roofline model executing multiple times the kernel with different arithmetic intensities. The Roofline Toolkit shows promising results. However, there are is some tuning to do in order to adapt it to some architectures. Calore et al. in [29] explain how they had to modify the algorithm of the Roofline Toolkit kernel in order to accommodate the limitations of the fused-multiply-and-add operation in Armv8. Moreover, the Roofline Toolkit struggles to approach the theoretical performance peaks. With the modifications proposed in [29], the authors go from $\sim$ 25% to $\sim$ 66% of the theoretical peak performance. It is a considerable gain, but still far from the peak.

In this thesis, I present a method to construct the roofline model from the results of the FPU μKernel and lmbench. The peak floating point performance F_p is the result labeled as "Total" of the FPU μKernel. The peak memory bandwidth of the system B_p is the result of the memory load bandwidth test. Both codes are run with a whole node of the system having one thread per core. The roofline models include ceiling curves for the different levels of caches as well as double precision SIMD and FPU units.

In Section 4.3 I showed that the sustained bandwidth on a given cache level has some variation. For simplicity, I define B_p of each cache level as the second highest measurement in that same level. This excludes measurements which correspond to a transition between cache levels.

Figure 4.6, 4.7, and 4.8 show the roofline model for one node of Dibona, MareNostrum4 and Power9. The x-axis represents the arithmetic intensity and the y-axis represents the sustainable performance. Solid lines represent ceilings in the roofline model. There are two floating point performance ceilings corresponding to the FPU and SIMD units. There are four memory ceilings corresponding to the three cache levels and the main memory. For clarity, I included dotted lines, which represent the floating point performance ceilings.

I analyze the roofline models from the point of view of when a machine becomes memory bound. A program or kernel that is memory bound is much more cumbersome than one that is bound by the floating point performance. We say that a machine that takes a lower arithmetic intensity than another to become memory bound, it becomes memory bound *slower*. Following this definition, and looking at the L1 cache ceilings, we say that Power9 becomes memory bound the slowest of the three machines, at an arithmetic intensity between 1/4 and 1/2. Dibona follows closely, becoming memory bound at 1/2. Lastly, MareNostrum4 is the fastest machine to become memory bound at an intensity of 1. Looking exclusively at the L1 cache, it seems that the theoretical peak performance of a memory bound program degrades faster on MareNostrum4 than in the other two clusters. Please note that I am referring to the trend in sustainable performance, not the actual performance, which is still higher in MareNostrum4.

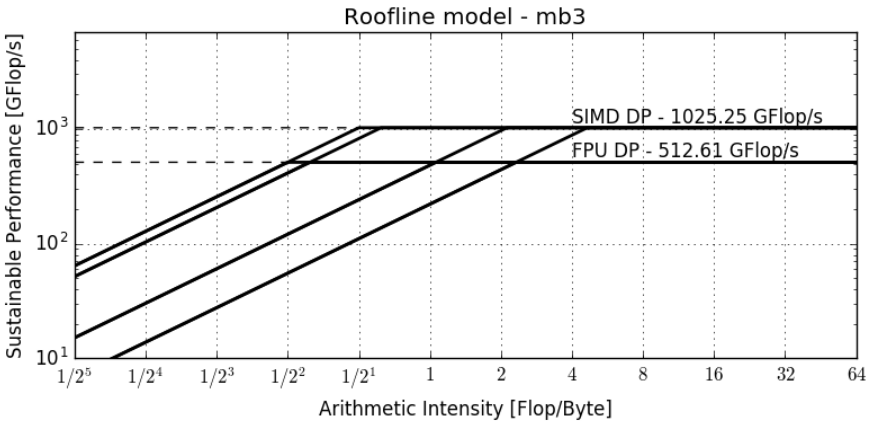

Figure 4.6: Roofline model in Dibona with double precision arithmetic

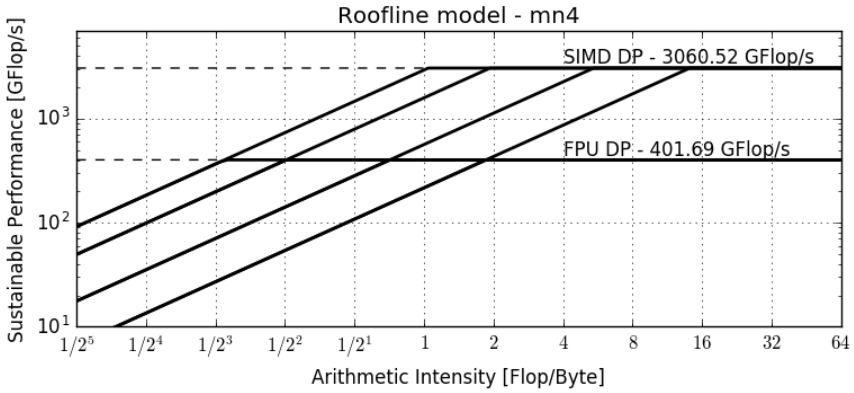

Figure 4.7: Roofline model in MareNostrum4 with double precision arithmetic

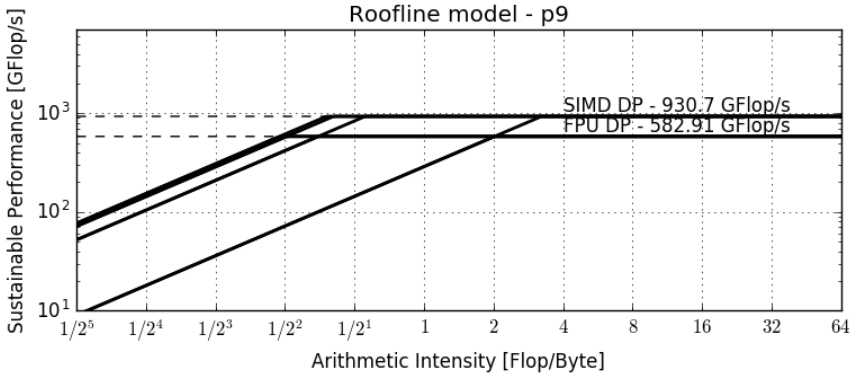

Figure 4.8: Roofline in Power9 with double precision arithmetic

We can interpret the gap between the L1 and L2, and L2 and L3 caches as the gain in theoretical peak performance when leveraging cache locality. In the case of Dibona, there seems to be a lower gain between L1 and L2 than in MareNostrum4. In Power9, there is little no no gain. This is because, as seen in Section 4.3, we measured that there is no memory bandwidth drop when jumping from L1 to L2 cache. In Dibona, the widest gap is from L2 to L3 cache. This may be due to the topology inside the chip.

Looking exclusively at the main memory ceilings, we find the same pattern as in the L1 ceilings. Power9 is the slowest to become memory bound, closely followed by Dibona and with MareNostrum4 being the fastest of the three. Most scientific applications have arithmetic intensities much lower than 1, as shown in Chapter 6. Based on the roofline model, a machine which becomes memory bound slower than another, has a theoretical peak performance which degrades slower. Please note the emphasis on *theoretical peak*. There are multiple factors which may make the sustained performance lower than the theoretical peak. The most important of them being the software environment. Depending on how the compiler leverages the system's resources, it yields a better or worse performance.

In conclusion, the roofline model defines a best-case scenario for a code or kernel with a given arithmetic intensity. This theoretical peak might not be reached in practice due to the system's software. However, if we measure both the theoretical peak and sustained performance, we have a way to determine how much room of improvement there is for a certain region of code. In Chapter 5 I explain how to evaluate the system's software environment to tied it back into the roofline model.

4.5 Memory hierarchy latency

lmbench has a test to measure memory load latency. I used this benchmark in Dibona and reported the results in [2]. I was unable to reproduce the results using the same tool. For this reason, I decided to write a simpler version of the test.

The test I wrote measures the memory hierarchy's load latency at different cache levels. The benchmark takes a size n, in elements, and a stride S as parameters. It constructs an array of n elements where each element has a size of 8 Bytes. Each element e_i in the array is a pointer to another element which is strided by S elements. For each pair of elements e_i and e_{i+S}, with $S < i < n$ and $0 < S < n$, the element e_{i+S} points to e_i. We define a chain of pointers as the sequence of elements that points to each other. The start of the sequence is the element with the highest index. The end of the sequence is the element with the lowest index. Figure 4.9 shows a chain of pointers in an array.

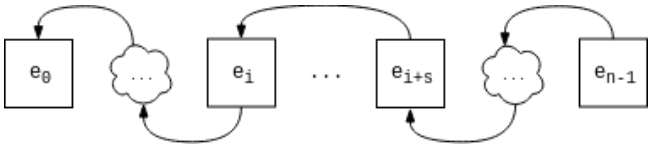

Figure 4.9: Memory latency test ring of pointers. Each element of the array points backwards S elements.

Once the array is constructed, the test traverses the array following the chain of pointers that ends in element zero. The traversal is done by dereferencing each pointer in the chain. A dereference is equivalent to a memory read. The number of memory reads in one traversal of the array is equal to $(n-1)/S$. To get the time per memory access, it would be enough to divide the total time of a traversal by the number of accesses. However, the resolution of the `gettimeofday` C system call is not enough for this measurement. For this reason, the program does multiple traversals of the array and reports the average access time.

One run of the program returns an average access time for a given size n and stride S. By keeping constant S and increasing n, the allocated array starts by filling the L1 cache, then the L2 cache and, finally, the L3 cache. If we run the program for multiple values of n, we are be able to plot the memory latency at each level of the memory hierarchy.

For my experiments, I used a sequence of input n which is more dense around sizes that trigger a jump up in the cache hierarchy. This means that I made more measurements with array sizes close to the capacity of a cache level and less measurements while the array is filling a cache partially. With this sequence, I get a better resolution to observe the increase in access time when going from one cache level to the next. I used strides $S \in 32, 64, 128, 256, 512$. Measurements with strides that are not powers of two produce inconsistent results.

Figure 4.10 shows the results of the memory latency micro-benchmark in Dibona. The x-axis represents the array size in KiB and the y-axis represents the access time in nanoseconds. Each line represents a series of runs with a fixed stride. The thick black vertical lines represent the size of the L1 and L2 caches.

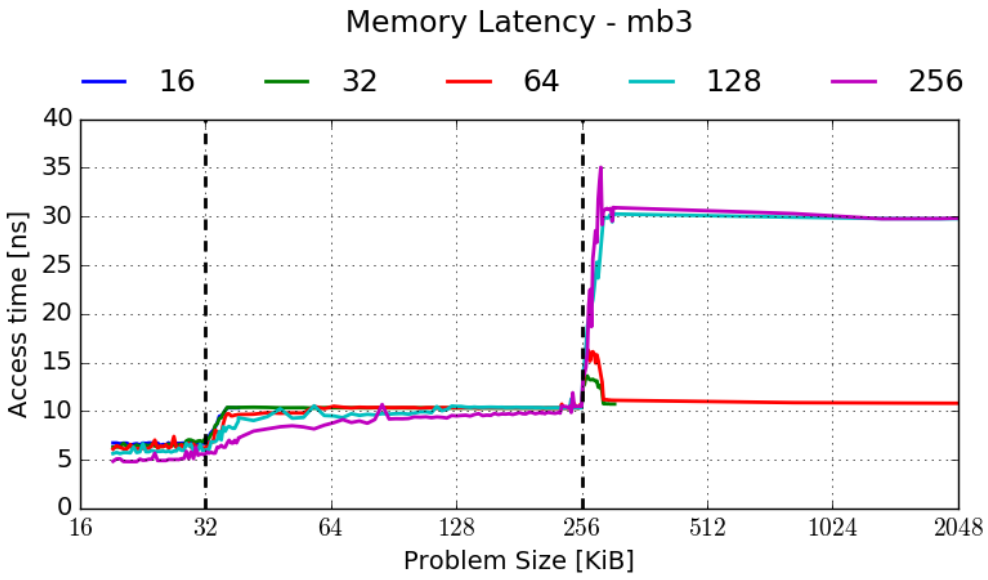

Figure 4.10: Memory latency test in Dibona

The vertical lines in the plot divide the x-axis into three regions which correspond to the different levels of cache. When the problem size fits in the L1 cache, the access time is 7ns. Moving up to the L2 cache, the access time increases to 10ns. The jump to L3 cache is clearly visible for $S \in 128, 256$ but not so much for $S \in 32, 64$. For big strides, the access time increases to 30ns, indicating that accesses are truly occurring at the L3 cache. However, for small strides, the access time increases a bit when the problem size is around 256KiB and then drops close to 10ns. I do not have a conclusive explanation for this behavior but I would suggest that it is an effect of the hardware prefetcher. The small strides are simple enough for the prefetcher to be trained and have a high accuracy.

Another observation that we can make by looking at Figure 4.10 is that big strides ($S \in 256$) have a lower access time during the L1 and L2 regions of the plot. I attribute this to not having enough time resolution in the measurements. The traversals of small arrays with such a big stride are too quick to time precisely.

Figure 4.11 shows the results of the memory latency micro-benchmark in MareNostrum4. As with Dibona, we see increasing access time when going up in the memory hierarchy. However, the results in MareNostrum4 are not as clear as with Dibona.

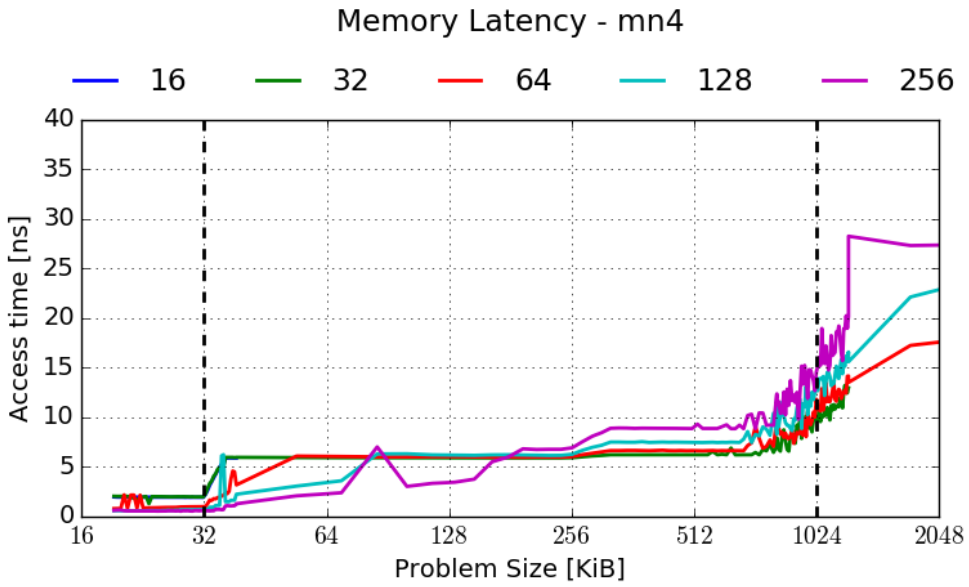

Figure 4.11: Memory latency test in MareNostrum4

The access time to the L1 cache for $S \in 16, 32$ is 2ns. The rest of strides in this range of sizes suffer from low clock resolution and are inconclusive. For the L2 cache, we observe an access time of 6.5ns for $S \in 16, 32$. At a problem size around 64KiB, we also see that measurements with $S \in 64, 128$ result in the same access time as with smaller strides. At the tail end of the L2 cache region, we can observe a plateau that gets higher for larger strides. The access time then slowly transitions to the L3 cache. This transition is much slower than in Dibona. Once the problem size has reached the L3 region, we see widely different access times depending on the stride: 17ns for $S = 64$, 25ns for $S = 128$, and 30ns for $S = 256$. The high variability on access times depending on the stride might be related on how well the caches manage data locality and how good the hardware prefetcher can predict the access pattern.

Figure 4.12 shows the results of the memory latency micro-benchmark in Power9. The plot shows three distinct plateaus which correspond to the three levels of cache in Power9. However, there seems to be some variability in the measurements. For the L1 cache, the access time is between 6 and 10ns. For the L2 cache, the access time is between 10 and 14ns. Lastly, for the L3 cache, the access time is between 16 and 18ns. For any given plateau in Figure 4.12 the access time is bimodal. I currently do not have an explanation for this behavior, though it might be due to the vastly different memory hierarchy in Power9 with respect to the other two machines. I will try to investigate further in a future work.

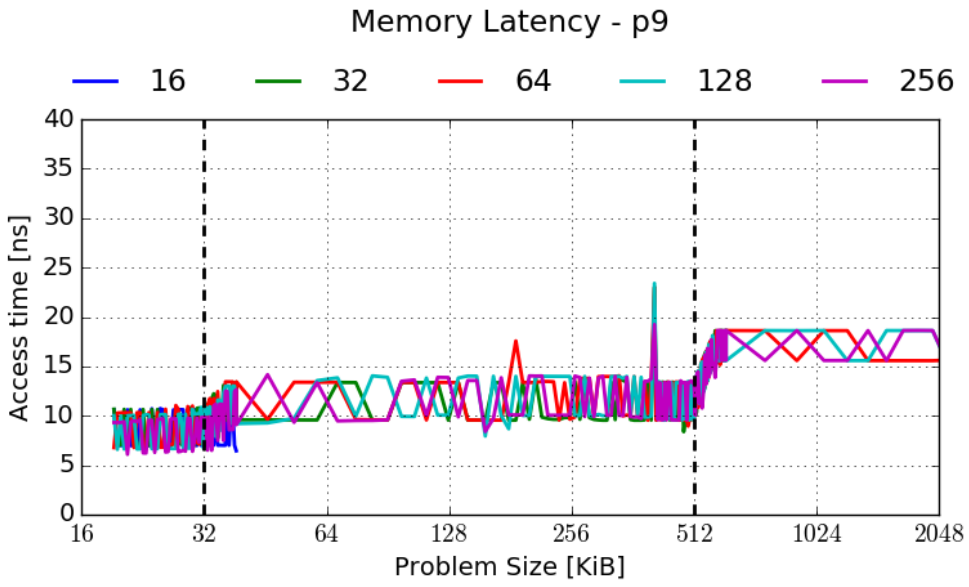

Figure 4.12: Memory latency test in Power9

In this section I have presented the preliminary results of a hand-made micro-benchmark which measures the read access latency to the different levels of the cache hierarchy. The results are promising but there are some effects in the measurements that are still not explained. In a future work, I will try to refine the code to better reflect the access time. The insight gained from these measurements would allow us to build a timing model of the memory hierarchy, which would be very useful for applications which are bound by the memory latency.

4.6 Infiniband read bandwidth

Dibona and Power9 have a communication network based on Infiniband. The provider of this technology, Mellanox, packages with its driver a set of tests to evaluate the performance of the network. In this section, I show the results of running the `ib_read_bw` utility. Please note that this utility is also present in MareNostrum4, but it runs through the Ethernet network, since MareNostrum4 does not have Infiniband.

This test needs a client and a server process. Both processes exchange messages of increasing size and the utility reports the achieved network bandwidth of each size. I placed the client and server processes in different nodes to measure the actual network bandwidth. Figure 4.13 shows the result of this test in both Dibona and Power9. The x-axis represents the message size and the y-axis represents the network bandwidth. I included a horizontal line which represents the theoretical peak bandwidth of the Infiniband EDR technology.

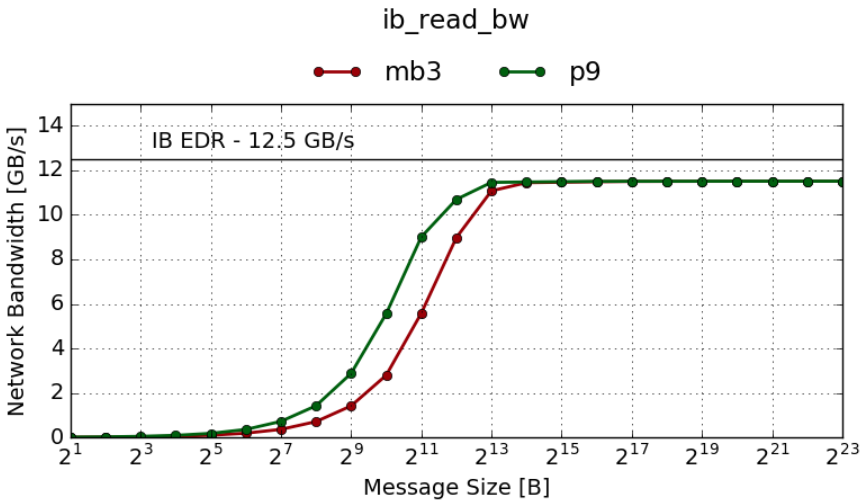

Figure 4.13: Infiniband read bandwidth in Dibona and Power9

Both clusters have a similar behavior when increasing the message size. The achieved network bandwidth follows a smooth curve that ramps up starting on messages bigger than 128 Bytes. The bandwidth saturates at messages greater than 8 KiB (2^{13} Bytes). It looks like the configuration in Power9 yields a better network bandwidth with messages between 128 B and 8 KiB. This could have an impact on performance for applications that are network bound and send messages in this range of sizes. The network bandwidth saturates at ~11.5 GB/s, which is 92% of the peak bandwidth. The missing 8% might be due to overheads in the driver or the physical network card. As an end user, we cannot delve deeper into the source of the inefficiencies.

It is important to note that all the clusters in this study are in production. Other users are using the communication network while I perform my measurements. I do not consider this a limitation of the model but rather a strength, since it emulates a real environment which a user is exposed to.

Chapter 5

System software

Experience without theory is blind, but theory without experience is mere intellectual play.

— "old Kantian maxim" as declared by the Society for the Advancement of General Systems Theory

In Section 2.2 I mentioned the importance of a strong software ecosystem on any given HPC system. Two of the most important software tools in an HPC cluster are the compiler and the communication library (i.e., MPI implementation). In this chapter, I present a methodology to study the optimizations that the compiler performs as well as the possible overheads introduced by the MPI library.

Sections 5.1, 5.2, and 5.3 study , through a set of small programs, how compiler optimization flags affect the autovectorization of the code. We use PAPI to measure the number of dynamic instructions and how much the compiler vectorized the code. For each machine and compiler, we compare for versions of each kernel depending on the data type (i.e., single or double precision); and if the code is plain scalar C or if it has been manually vectorized using intrinsics. The two compiler kernels I report in this thesis implement a multiplication `t = a*b` and a fused multiply and add `t = a*b + c`, which represent a common pattern of operations in HPC.

All the compiler kernels follow a similar structure. First, we allocate two arrays of N elements. N is defined at compile time. We then initialize the arrays with random values. Next, we traverse the arrays calling the kernel on each iteration. Let vl the length, in elements, of a vector register. We choose a value for N which is *i)* big enough to hide the overhead of the PAPI instrumentation; and *ii)* $N\%vl = 0$ so there are no tail elements when vectorizing. For this experiment, we chose $N = 8192$.

The third compiler kernel is a stencil code extracted from the miniAMR benchmark. It is a 27 point stencil kernel written in C. I study how much the compilers are able to autovectorize depending on how much the code exposes data reuse.

Lastly, I conclude the section by showing results of the OSU benchmark, which evaluates the MPI communication library. The implementation the communication library could cause some overheads over the physical implementation of the network. To better understand these overheads, we should compare the results in this section to the one found in Section 4.6

5.1 Multiply Kernel

Listing 5.1 shows the multiply compiler kernel. The first function is the scalar version. The second function is an example of manual vectorization using Intel's AVX512 vector extension. Notice the `__restrict__` clause that serves as a hint for the compiler, indicating that pointers `a` and `b` are not aliases and there is no need to check it. The functions are small enough that the compiler is able to inline them in the main routine.

```
inline void mul(double* __restrict__ a, double* __restrict__ b) {
  double va = *a;
  double vb = *b;
  double vt = va * vb;
  *a = vt;
}

inline void mul(double* __restrict__ a, double* __restrict__ b) {
  __m512d va = _mm512_load_pd(a);
  __m512d vb = _mm512_load_pd(b);
```

```
    __m512d vt = _mm512_mul_pd(va, vb);
    _mm512_store_pd(a, vt);
}
```

Listing 5.1: scalar and AVX512 version of the mul compiler kernel

Dibona

In Dibona, the available compilers are GCC and the Arm HPC Compiler. I compiled the scalar and vectorized versions of the kernel using both compilers. Listings 5.2 show 5.3 show the dissasembled binaries of the multiply kernel compiled with GCC and double precision. For brevity, I only included the part of the code that is measured. The instructions highlighted with an asterisk represent the main operation of the kernel (i.e., multiplication).

```
    bl      401048 <start_measurements>
    mov     x0, #0x0
    ldr     q0, [x19,x0]
    ldr     q1, [x20,x0]
*   fmul    v0.2d, v0.2d, v1.2d
    str     q0, [x19,x0]
    add     x0, x0, #0x10
    cmp     x0, #0x10, lsl #12
    b.ne    400c38 <main+0x158>
    ldr     w0, [sp,#92]
    mov     x1, x25
    bl      4010a8 <end_measurements>
```

Listing 5.2: Scalar mul kernel in Dibona with GCC

```
    bl      401048 <start_measurements>
    nop
    ldr     q0, [x19]
    ldr     q1, [x20],#16
*   fmul    v0.2d, v0.2d, v1.2d
    str     q0, [x19],#16
    cmp     x19, x21
    b.ne    400c38 <main+0x158>
    ldr     w0, [sp,#108]
    mov     x1, x28
    bl      4010a8 <end_measurements>
```

Listing 5.3: Vectorized mul kernel in Dibona with GCC

The two binaries are almost identical. The GCC compiler is able to autovectorize the multiply operation as well as the manual vectorization. However, Listing 5.2 has an extra instruction: `add x0, x0, #0x10` that prepares the operands of the loop check. This extra instruction in the binary makes it so the autovectorized version of the kernel performs more instructions during execution than the manually vectorized version.

Listing 5.4 shows the dissasembled binary compiled with the Arm HPC Compiler. I only include the binary of the scalar version because the manually vectorized version because there is no meaningful difference with the one compiled with GCC.

```
    bl      400e3c <start_measurements>
    mov     x8, xzr
    add     x9, x22, x8
    add     x10, x23, x8
    ldp     q0, q1, [x9]
    ldp     q2, q3, [x10]
    add     x8, x8, #0x20
    cmp     x8, #0x10, lsl #12
*   fmul    v0.2d, v2.2d, v0.2d
*   fmul    v1.2d, v3.2d, v1.2d
    stp     q0, q1, [x9]
    b.ne    4010e8 <main+0x144>
    ldr     w0, [sp, #12]
    mov     x1, x20
    bl      400ea0 <end_measurements>
```

Listing 5.4: Scalar mul kernel in Dibona using the Arm HPC Compiler

The most noticeable feature of the binary produced by the Arm HPC Compiler is that it has unrolled the loop by a factor of two. We observe that there are two `fmul` instructions per loop iteration. Right after the multiply instructions, we find a store-pair `stp` instruction, which stores two SIMD registers with a single instruction. By unrolling the loop and using the `stp` instruction, the Arm HPC Compiler reduces the total number of executed instructions. Please keep in mind that the `stp` reduces the total number of load instructions, but it may not benefit the overall performance of the kernel, since the amount of data sent to the memory is the same as when using two regular store instructions.

Figure 5.1, and 5.2 show the number of total and vector instructions of each execution. The x-axis represents different versions of the kernel. The first letter of the label indicates the precision: d for double and s for single. The second letter of the label indicates if the original code is scalar (s) or manually vectorized (v). The y-axis represents the value read from the PAPI counters. Lower is better.

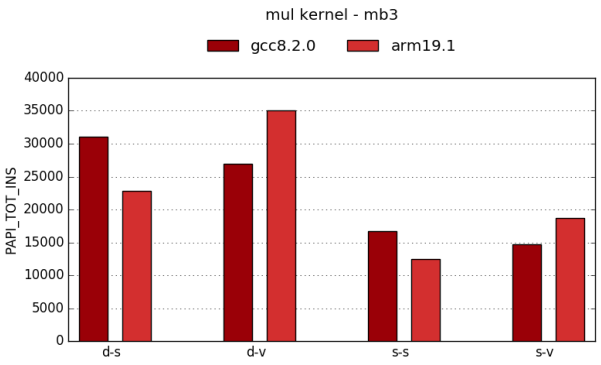

Figure 5.1: Total inst. of mul kernel in Dibona

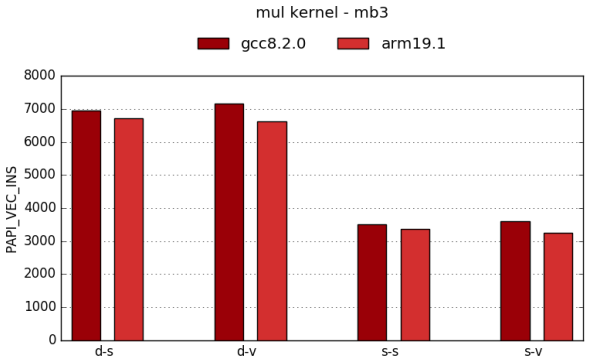

Figure 5.2: Vector inst. of mul kernel in Dibona

In the previous listings, we observed that the GCC scalar binary had one more static instruction that the manually vectorized one. This extra instruction results on $\times 1.15$ higher dynamic instruction count. In contrast, the scalar version compiled with the Arm HPC Compiler has an extra instruction, but it unrolls the loop by a factor of two, which results in a lower dynamic instruction count.

One SIMD register in Dibona can hold four single precision elements and two double precision elements. With $N = 8192$, we expect to execute $V = N/vl$ vector arithmetic instructions. For single precision $V = 2048$, and for double precision 4096. However, I measured $\times 1.5$ more vector instructions than expected. I currently do not have an explanation for this result.

MareNostrum4

In MareNostrum4, the available compilers are GCC and the Intel Compiler. Listings 5.5, 5.6 show the dissasembled binaries of the kernel compiled with GCC and with double precision.

```
  callq  400f00 <start_measurements>
  mov    -0x38(%rbp),%rcx
  mov    -0x40(%rbp),%rdx
  lea    0x20(%rdx),%rax
  cmp    %rax,%rcx
  jae    400b15 <main+0x1a5>
  lea    0x20(%rcx),%rax
  cmp    %rax,%rdx
  jb     400ba6 <main+0x236>
  xor    %eax,%eax
  nopw   0x0(%rax,%rax,1)

* vmovupd (%rdx,%rax,1),%ymm1
* vmulpd (%rcx,%rax,1),%ymm1,%ymm0
* vmovupd %ymm0,(%rdx,%rax,1)
  add    $0x20,%rax
  cmp    $0x10000,%rax
  jne    400b20 <main+0x1b0>
  vzeroupper
  mov    %r13,%rsi
  mov    -0x44(%rbp),%edi
  callq  400f50 <end_measurements>
```

Listing 5.5: Scalar mul kernel in MareNostrum4 with GCC

```
  callq  400ec0 <start_measurements>
  xor    %eax,%eax
  nopl   0x0(%rax)
  mov    -0x40(%rbp),%rdx
  add    %rax,%rdx
  mov    -0x38(%rbp),%rcx
* vmovapd (%rdx),%zmm1
* vmulpd (%rcx,%rax,1),%zmm1,%zmm0
* vmovapd %zmm0,(%rdx)
  add    $0x40,%rax
  cmp    $0x10000,%rax
  jne    400b00 <main+0x190>
  mov    %r13,%rsi
  mov    -0x44(%rbp),%edi
  vzeroupper
  callq  400f10 <end_measurements>
```

Listing 5.6: Vectorized mul kernel in MareNostrum4 with GCC

In MareNostrum4, the binary of autovectorized version of the kernel produced by GCC is much more complex than the manually vectorized. The most relevant difference is that the autovectorized binary does not leverage the AVX512 instructions, which use SIMD registers of 512 bits (`zmm` registers). Even when using the optimization flags provided by the system administrators[1], the autovectorized kernel uses `ymm` SIMD registers, which are 256 bits wide.

Another important difference between the autovectorized and the manually vectorized versions of the kernel is the amount of instructions surrounding the actual multiply operation. In Listing 5.6 we see that there are a lot more instructions to prepare the operands and perform bookkeeping of the loop iterations in comparison with the manually vectorized binary.

Listing 5.7 shows the dissasembled binary of the multiply kernel compiled with the Intel Compiler and double precision elements.

```
  callq   401ab0 <start_measurements>
  mov     (%rsp),%r9
  mov     %r9,%r8
  and     $0x1f,%r8
  mov     0x8(%rsp),%r10
  test    %r8d,%r8d
  je      401e59 <main+0x209>
  test    $0x7,%r8d
  jne     40204c <main+0x3fc>
        [...]
* vmovsd (%r9,%rax,8),%xmm0
* vmulsd (%r10,%rax,8),%xmm0,%xmm1
* vmovsd %xmm1,(%r9,%rax,8)
  inc     %rax
  cmp     %rcx,%rax
  jb      401e3d <main+0x1ed>
  jmp     401e5b <main+0x20b>
        [...]
* vmovupd (%r9,%rcx,8),%ymm0
* vmovupd 0x20(%r10,%rcx,8),%ymm2
* vmovupd 0x40(%r10,%rcx,8),%ymm4
* vmovupd 0x60(%r10,%rcx,8),%ymm6
* vmulpd (%r10,%rcx,8),%ymm0,%ymm1
* vmulpd 0x20(%r9,%rcx,8),%ymm2,%ymm3
* vmulpd 0x40(%r9,%rcx,8),%ymm4,%ymm5
* vmulpd 0x60(%r9,%rcx,8),%ymm6,%ymm7
        [...]
  callq   401b00 <end_measurements>
```

Listing 5.7: Scalar mul kernel in MareNostrum4 with the Intel Compiler

The first notable difference in the binaries generated by the Intel Compiler is their size. The binaries produced by GCC weigh 18 kB while the Intel Compiler generates binaries of 104 kB. The Intel Compiler produces a binary with multiple paths that are specialized to certain scenarios. For example, the scalar version of the multiply kernel compiled with the Intel Compiler has a path that uses 128 bit SIMD registers (`xmm` registers) and another path that uses 256 bit registers (`ymm` registers). The path to follow is determined at runtime depending on the number of elements left in the vector.

Figure 5.3 shows that the static size of the binary does not correlate to the final number of instructions executed. For all versions of the multiply kernel, GCC executes more instructions than the Intel Compiler. This is because of the multiple specialized paths that the Intel Compiler produces. Loop unrolling justifies larger binaries that execute less instructions compared to the unoptimized binary. We see in Listing 5.7 up to four AVX instructions per iteration, which amounts to a total of 128 elements.

Looking at Figure 5.4, we see that the number of vector instructions executed is identical for both compilers. In contrast to the results obtained in Dibona, the amount of vector instructions measured is around the expected.

[1] https://www.bsc.es/user-support/mn4.php

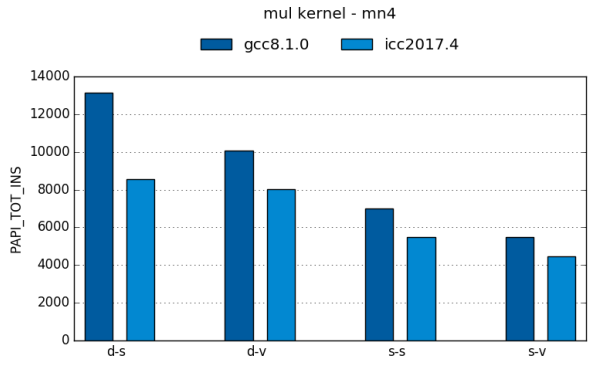

Figure 5.3: Total number of instructions of mul kernel in MareNostrum4

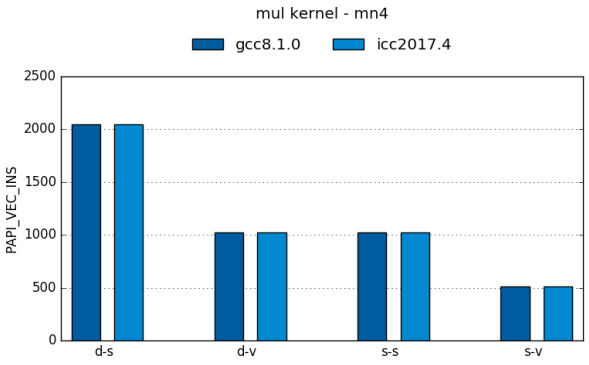

Figure 5.4: Vector instructions of mul kernel in MareNostrum4

Power9

In Power9, the available compilers are GCC, the PGI compiler, and the XL compiler (provided by IBM). The Power9 architecture has two vector extensions: AltiVec and VSX. Similar on how Intel's AVX and AVX512 build one on top of the other, VSX is an extension of AltiVec. The PGI compiler does not support vector intrinsics. The XL compiler does not support double precision VSX intrinsics. So for these two compilers I could not produce a manually vectorized code like with the other machines I presented in this section.

Listings 5.8 and 5.10 show the dissasambled binary of the multiply kernel compiled in Power9 with GCC and double precision elments. The autovectorized binary has more instructions that perform checks and brach outside the loop. We would expect that the autovectorized binary executes more instructions compared to the manually vectorized one. Looking at the multiply operation, we observe that both binaries use the same VSX instruction, `xvmuldp`. We can say that, for this kernel, the autovectorization done by GCC is on par with the manual vectorization.

```
  bl      <start_measurements+0x8>
  nop
  ld      r8,96(r1)
  ld      r10,104(r1)
  li      r9,1
  addi    r7,r8,16
  cmpld   cr7,r10,r7
  bge     cr7,10000dd8 <main+0x1f8>
  addi    r7,r10,16
  cmpld   cr7,r8,r7
  isel    r9,0,r9,28
  cmpwi   cr7,r9,0
  beq     cr7,10000ea4 <main+0x2c4>
  li      r7,4096
  li      r9,0
  mtctr   r7
  nop
  nop
  ori     r2,r2,0
  lxvx    vs0,r10,r9
  lxvx    vs12,r8,r9
* xvmuldp  vs0,vs0,vs12
  stxvx   vs0,r10,r9
  addi    r9,r9,16
  bdnz    10000df0 <main+0x210>
  lwa     r3,112(r1)
  mr      r4,r27
  bl      <end_measurements+0x8>
```

Listing 5.8: Scalar mul kernel in Power9 with GCC

```
  bl      <start_measurements+0x8>
  nop
  li      r9,4096
  li      r10,0
  mtctr   r9
  nop
  ori     r2,r2,0
  ld      r9,104(r1)
  ld      r8,96(r1)
  add     r9,r9,r10
  lvx     v0,r8,r10
  addi    r10,r10,16
  rldicr  r9,r9,0,59
  lxv     vs0,0(r9)
* xvmuldp  vs0,vs0,vs32
  stxv    vs0,0(r9)
  bdnz    10000dc0 <main+0x1e0>
  lwa     r3,112(r1)
  mr      r4,r27
  bl      <end_measurements+0x8>
```

Listing 5.9: Vectorized mul kernel in Power9 with GCC

Listings 5.10 and 5.11 show the dissasembled binaries of the multiply kernel compiled with PGI and XL. Notice how both binaries are significantly different to the ones produced by GCC.

The PGI compiler produces a binary that uses the `xvmuldp` instruction, just like GCC. However, PGI has also unrolled the loop by a factor of eight. There seem to be some instructions at the begining of the loop that perform checks and branch outside the loop in case a certain condition is met.

In contrast, the XL compiler is unable to leverage the VSX vector extension. We observe that the binary uses the `xsmuldp` which operates half of the elements as the `xvmuldp` instruction. The XL compiler unrolls the loop by a factor of two, which decreases the number of executed instructions by a bit. But overall, we expect that the binary produced by the XL will perform worse than the ones produced by GCC and PGI.

```
   bl      <start_measurements+0x8>
   nop
   ld      r3,104(r1)
   ld      r4,96(r1)
   li      r5,0
   li      r6,8193
   nop
   nop
   ori     r2,r2,0
   lxvx    vs0,r3,r5
   lxvx    vs1,r4,r5
   add     r7,r3,r5
   add     r8,r4,r5
   addi    r6,r6,-16
   cmplwi  r6,4
*  xvmuldp vs0,vs0,vs1
   stxvx   vs0,r3,r5
   addi    r5,r5,128
   lxv     vs0,16(r7)
   lxv     vs1,16(r8)
*  xvmuldp vs0,vs0,vs1
   stxv    vs0,16(r7)
         [...]
   bgt     10001790 <main+0x258>
   lwa     r3,116(r1)
   mr      r4,r28
   bl      <end_measurements+0x8>
```

Listing 5.10: Scalar mul kernel in Power9 with PGI

```
   bl      <start_measurements+0x8>
   nop
   ld      r3,120(r1)
   ld      r4,128(r1)
   li      r0,8191
   addi    r3,r3,-8
   addi    r4,r4,-8
   mtctr   r0
   addi    r0,r4,8
   addi    r5,r3,8
   addi    r6,r3,8
   addi    r5,r4,8
   lfd     f0,8(r3)
   lfd     f1,8(r4)
*  xsmuldp vs10,vs0,vs1
   nop
   stfd    f10,0(r6)
   addi    r0,r5,8
   addi    r3,r6,8
   lfd     f2,8(r6)
   lfd     f3,8(r5)
*  xsmuldp vs10,vs2,vs3
   addi    r5,r5,8
   addi    r6,r6,8
   bdnz    10001680 <main+0x220>
   stfd    f10,0(r6)
   lwa     r3,96(r1)
   mr      r4,r26
   bl      <end_measurements+0x8>
```

Listing 5.11: Scalar mul kernel in Power9 with XL

Figure 5.5 shows the total number of executed instructions of each version of the multiply kernel in Power9. Recall that the double precision version cannot be manually vectorized with PGI and XL since they do not support the VSX intrinsics. The PGI compiler also does not support the intrinsics of the single precision version. In Figure 5.5, I denote the unimplemented versions of the kernel with no column and the number 0.

As expected, the binaries produces by the XL compiler execute more instructions that the binaries produced by the other two compilers. On the other hand, GCC and PGI are in the same range of dynamic instructions. It is unclear which one of the two will execute less instructions.

Figure 5.6 shows the number of vector floating point operations executed by each version of the kernel. We observe very unexpected measurements. On the one hand, the number of operations is the same for the GCC and PGI compiler in the double precision autovectorized version. On the other hand, GCC produces a binary which executes half of the vector operations for the single precision of the autovectorized version. This is because GCC is able to use VSX instructions to leverage the whole SIMD registers, while PGI does not.

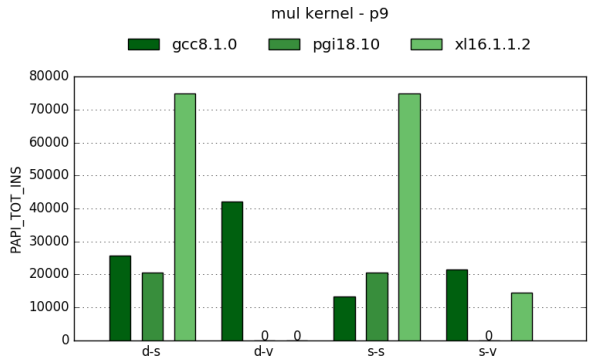

Figure 5.5: Total inst. of mul kernel in Power9

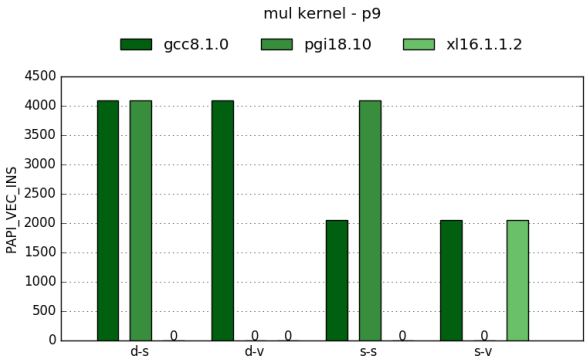

Figure 5.6: Vector inst. of mul kernel in Power9

5.2 FMA Kernel

All versions of the fused-multiply-add kernel in all machines with their respective compilers yield results that are coherent with the observations we made with the multiply kernel. The compilers use similar optimization strategies, compared with the multiply kernel, with the most notable difference being that the kernel has now two floating point operations: multiply and add. However, all ISAs include a fused-multiply-add instruction which is used by the compilers when they autovectorize the code.

5.3 Stencil

In this section, I present the measurements of a stencil kernel extracted from miniAMR[44]. Stencil computations are a relevant in the field of HPC and so we want to evaluate how the compiler is able to optimize the access patterns. The program takes a problem size S as an input parameter to construct a three-dimensional array of S^3 double precision floating point elements. Each element of the array is initialized with a random value. The program allocates a second array of S^3 elements to store the results of the kernel. The stencil kernel traverses the input array with three nested `for` loops that access, respectively, the x-axis with the induction variable i, the y-axis with the induction variable j, and the z-axis with the induction variable k. For each element (i, j, k) of the input array, the kernel computes the average of all adjacent cells plus the given element. We consider a 27 element connectivity. The result is stored in element (i, j, k) of the result array. Figure 5.7 shows *i)* the orientation of the x-axis, y-axis, and z-axis of the matrix; and *ii)* a given element (i, j, k), in blue, surrounded by its adjacent cells, in red.

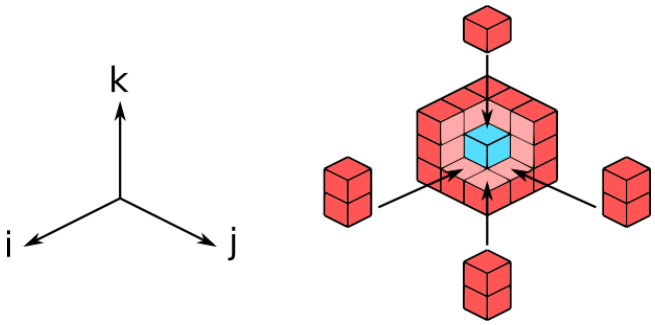

Figure 5.7: Left: Orientation of axes of the matrix. Right: Adjacent cells of a given element (i, j, k)

I wrote two extended versions of the original stencil kernel. The first version, labeled `unroll2`, unrolls the innermost loop (i.e., the one that traverses the matrix in the z-axis) by a factor of two. The objective of this version is to expose to the compiler more opportunities to vectorize arithmetic operations, since there are more sums per iteration of the k loop.

The second version, labeled `reuse`, starts from `unroll2` and is built upon the knowledge that two elements (i, j, k) and $(i, j, k + 1)$ have adjacent cells in common. The `reuse` implementation of the kernel reuses the data of the common adjacent cells to reduce the memory pressure of each iteration. Figure 5.8 shows the two elements accessed, in blue, and their adjacent cells, in red, per iteration of the k loop in the `unroll2` and `reuse` implementations of the kernel.

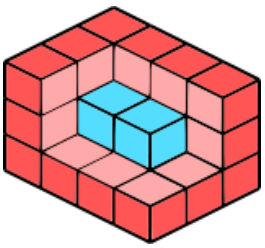

Figure 5.8: Elements (i, j, k) and $(i, j, k + 1)$ of the matrix, in blue, and their adjacent cells, in red

The general structure of the program is as follows. First, allocate memory for three arrays (A, B, and C) of S^3 double precision elements. Initialize each element of A with random values. Run the original version of the stencil kernel with A and B as the input and result arrays, respectively. Run the `unroll2` version of the stencil kernel with A and C as the input and result arrays, respectively, and validate the result by comparing B and C. Run the `reuse` version of the stencil kernel with A and C as the input and result arrays, respectively, and validate the result by comparing B and C. Deallocate arrays A, B, and C.

I used PAPI to measure the amount of cycles, issued instructions and vector instructions of each invocation of the stencil kernel. I fixed the problem size S so that the total memory used by the program would be, at least, 80% of the total memory of one node. Let M the total amount of memory of one node in Bytes. The minimum problem size S of the stencil kernel is:

$$S = \left\lceil \sqrt[3]{\frac{M \times 0.8}{8 \times 3}} \right\rceil$$

For the rest of this section, I present the measurements of the program on each machine. For each machine, I run the program three times and measured that the standard deviation of the measurements is always under 2% of the average value across all executions. Since the results have almost no variability, I present only the measurements of the first run on each machine.

In addition to the measurements of the PAPI counters, I also present the autovectorization ratio. This metric allows us to quantify the amount of autovectorization that each compiler is able to do on each machine. The autovectorization ratio is defined as the ratio between the number of vector instruction divided the total number of instructions.

Dibona

In Dibona, the Arm HPC Compiler is unable to autovectorize at all regardless of the version of the kernel. It seems that the memory access pattern of the stencil is too complex for the autovectorizer. On the other hand, GCC autovectorizes all of the versions. The autovectorization ratio of the reference version compiled with GCC is 0.57. It increases when implementing the `reuse` and `unroll` optimizations, getting up to 0.69.

Figure 5.9 shows the total cycles for each version of the stencil kernel with both compilers in Dibona. It is apparent that GCC has an edge in cycles over the Arm HPC Compiler. This is because GCC compiler executes less instructions than the Arm HPC Compiler thanks to the autovectorization. For the reference version, GCC has a $\times 2$ speedup over the Arm HPC Compiler. For the other two versions, GCC has a $\times 1.70$ speedup.

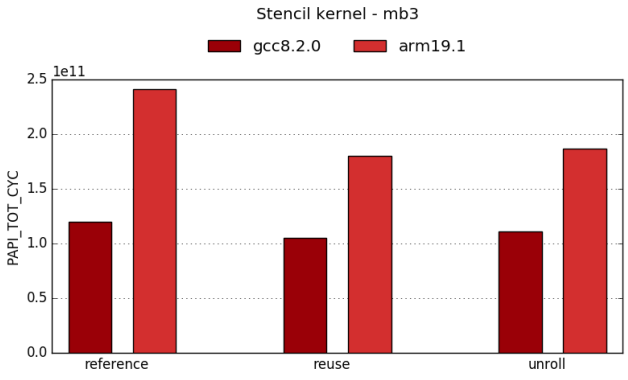

Figure 5.9: Total cycles of stencil kernel in Dibona

MareNostrum4

In MareNostrum4, both compilers are able to autovectorize the code. Figure 5.10 shows the total cycles of each version of the stencil kernel. Figure 5.11 shows the total number of instructions executed by each version of the stencil kernel.

For the reference version, the binary produced by the Intel Compiler executes less instructions than the one from GCC. In turn, the binary generated by the Intel Compiler takes less cycles to complete the kernel. However, for the `reuse` and `unroll` versions of the kernel, the binary produced by Intel Compiler executes way more instructions compared to the one from GCC. In the worst case, the binary generated by the Intel Compiler executes more than three times as many instructions as the binary from GCC. This huge difference in dynamic instruction count results on the binary produced by GCC taking significantly less cycles to complete in comparison to the one from the Intel Compiler. Looking at Figure 5.10 and 5.11, it is apparent that the number of cycles is strongly correlated to the number of executed instructions.

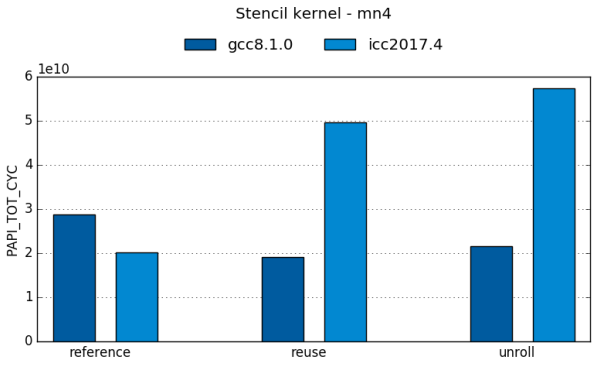

Figure 5.10: Total cycles of stencil kernel in MareNostrum4

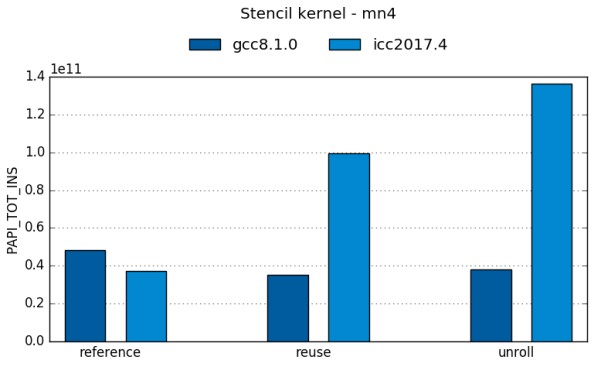

Figure 5.11: Total inst. of stencil kernel in MareNostrum4

Power9

In Power9, only GCC and PGI are able to autovectorize part of the stencil kernel. Figure 5.12 shows the total cycles of each version of the stencil kernel. Figure 5.13 shows the total number of instructions executed by each version of the stencil kernel.

For the reference version, all compilers have a very similar cycle count. It is not clear which one could yield a better performance. Moving on to the `reuse` version, we observe that the binary produced by GCC takes less cycles to complete while the binaries generated by PGI and XL do not benefit from this code modification. Lastly, we observe that the `unroll` version has an increase of cycle count for all compilers.

Looking at the instruction count in Figure 5.13, we see that the `unroll` and `reuse` versions execute a higher number of instructions when compiled with PGI and XL. This is because the PGI compiler does not vectorize when applying the code modifications and the XL compiler does not vectorize at all. On the other hand, GCC decreases the number of instructions when going from the reference version to `reuse` but there is no gain from the `reuse` to the `unroll` versions.

By observing both Figure 5.12 and 5.13, we can conclude that the increase in cycle count when running the `unroll` version is not due to the number of instructions executed. In contrast to what we saw in MareNostrum4, there is not a strong correlation between the number of cycles and the number of executed instructions. Actually, the number of cycles increases in the `unroll` version of the kernel due to code replication, which leads to lower instruction cache locality.

5.4 OSU Benchmarks

In Section 4.6 I presented the network bandwidth in Dibona and Power9 measured with the tools provided by the manufacturer, Mellanox. I showed that the results from the `ib_read_bw` test approach the theoretical peak network bandwidth. These utilities have the minimum software components to communicate through the IB interconnect. However, scientific applications use, at least, one more software component to communicate processes. This added layer is the MPI communication library. Depending on the MPI flavor (i.e., OpenMPI, mpich, Intel MPI, etc.) and how it has been configured by the system administrator, the final user might not be able to leverage the network bandwidth to its fullest due to overheads.

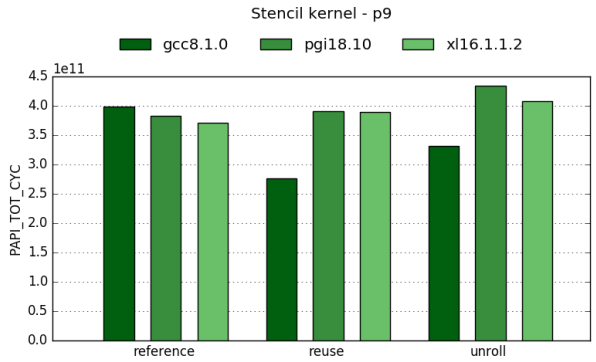

Figure 5.12: Total cycles of stencil kernel in Power9

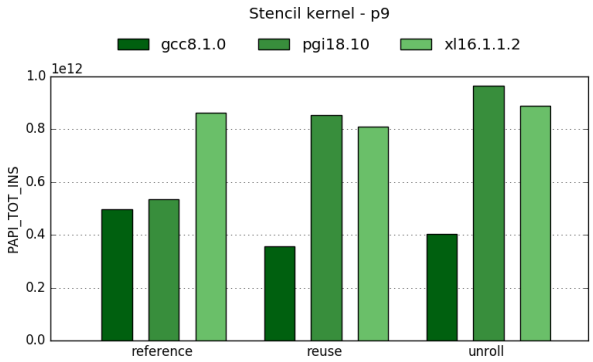

Figure 5.13: Total inst. of stencil kernel in Power9

In this section, I present the network bandwidth of Dibona, MareNostrum4, and Power9 measured with the OSU benchmarks [45]. These are a collection of small MPI programs that measure latency and bandwidth of different MPI primitives. The `osu_bw` test, under `pt2pt` collection of test is an MPI program with a similar behavior to the `ib_read_bw`. Two processes exchange point-to-point messages of a given size with blocking MPI primitives. Figure 5.14 shows the achieved throughput, as a function of the message size of the communication. All points represent the average value of 100 repetitions of the communication.

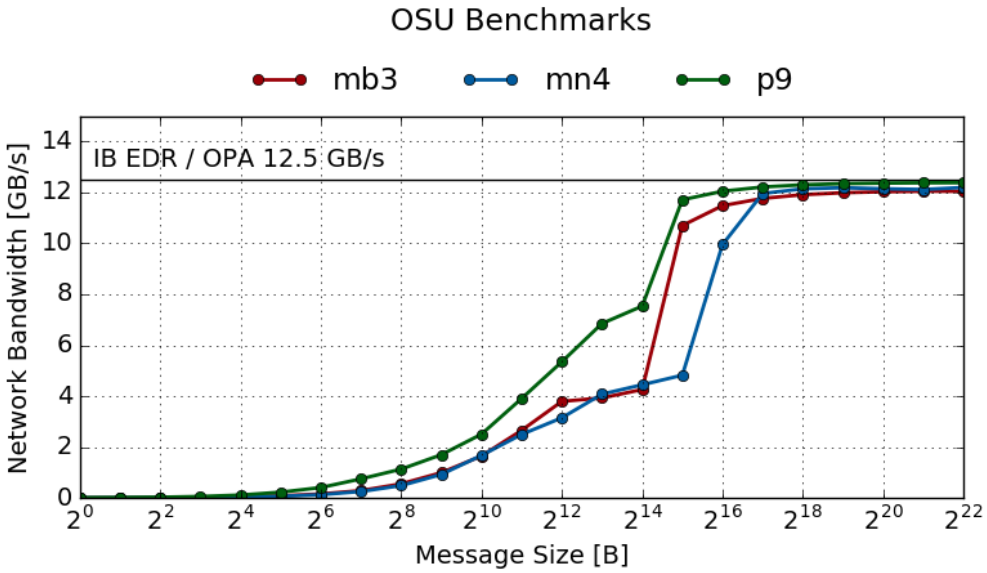

Figure 5.14: Bandwidth between two processes in different nodes

Both networks approach the theoretical peak as the message size increases (∼95%). It seems that OPA is consistently achieving a better bandwidth than IB with message sizes over 256 KiB. The difference in bandwidth is also very noticeable at message sizes around 4 KiB and 8 KiB, where OPA almost doubles IB. It seems that measured bandwidth of OSU in Dibona stalls around 8 and 16 KiB but then shoots up to 10 GB/s for larger message sizes. This behavior is consistent throughout multiple pairs of nodes and between executions. We verified that this behavior disappears if we measure the bandwidth with the `ib_read_bw` tool by Mellanox. As this tool exchanges data using the raw network protocol, we can only conclude that the "valley" appearing in Figure 5.14 is caused by the OpenMPI configuration deployed by Bull/ATOS on the Dibona cluster at the moment of the tests.

Chapter 6

Scientific applications - Alya

All our knowledge begins with the senses, proceeds then to the understanding, and ends with reason. There is nothing higher than reason.

— Immanuel Kant

6.1 Application characterization

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 [46]. There exist mainly two numerical methods to solve partial differential equations on unstructured meshes, namely the finite volume and the finite element method [47]. 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 [48].

Alya is fully parallelized with MPI. The mesh partitioning is achieved using METIS [49], minimizing the number of neighbors.

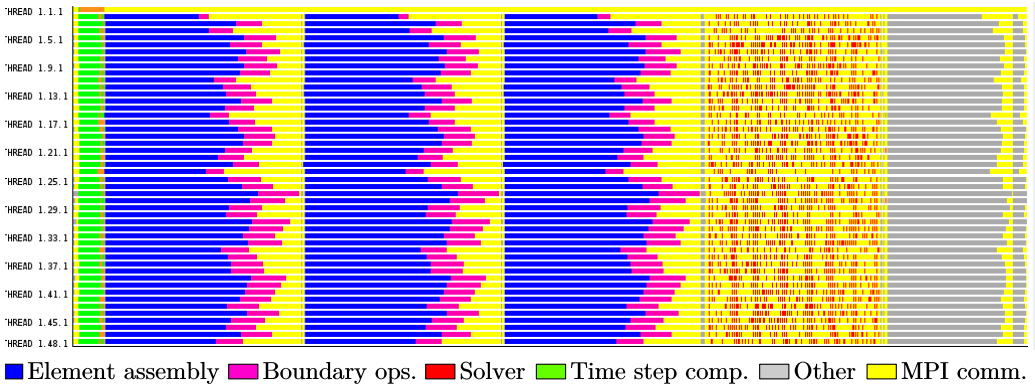

Figure 6.1: Timeline of one time step of the simulation

To identify the different computational phases of Alya we used Extrae to obtain a trace with 48 MPI processes and visualize it using Paraver. In Figure 6.1 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 time 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 [50].

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.

6.2 Compiler comparison

As mentioned in Chapter 5, 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 are 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), IPC and autovectorization.

Methodology

Table 6.1 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.

Table 6.1: 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`			

We used Extrae to collect data from the PAPI library during the execution of Alya and generate a trace. We then used Paraver 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 phase p of Alya introduced in Section 6.1, 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 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:

$$f_p = \sum_{j=1}^{P}\sum_{i=1}^{B} f_{i,j} \qquad m_p = \sum_{j=1}^{P}\sum_{i=1}^{B} m_{i,j} \qquad t_p = \max_{j=1}^{P}\sum_{i=1}^{B} t_{i,j} \tag{6.1}$$

Since the architectures under study offer different sets of hardware counters, we measure f and m using the PAPI events as described in Table 6.1.

Since the TX2 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 TX2 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.

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 I presented in Section 4.4. Figure 6.2 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 6.2 represents a theoretical room of improvement (please note that both axes are represented in a logarithmic scale).

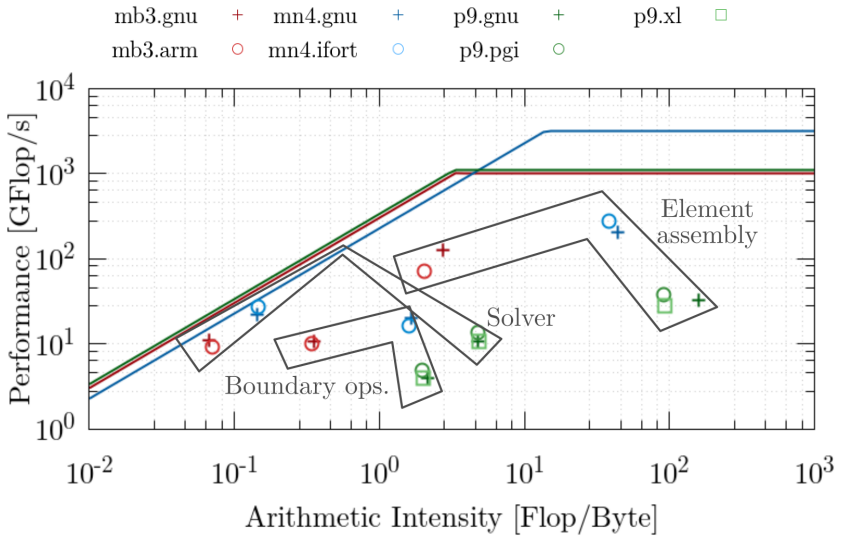

Figure 6.2: Roofline model of Dibona, MareNostrum4 and Power9 and measured performance per phase

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 6.2), 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 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 6.2 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.

Instructions Per Cycle

The IPC is a relevant metric to understand how busy a CPU is, 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 6.3 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 6.3 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.

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.

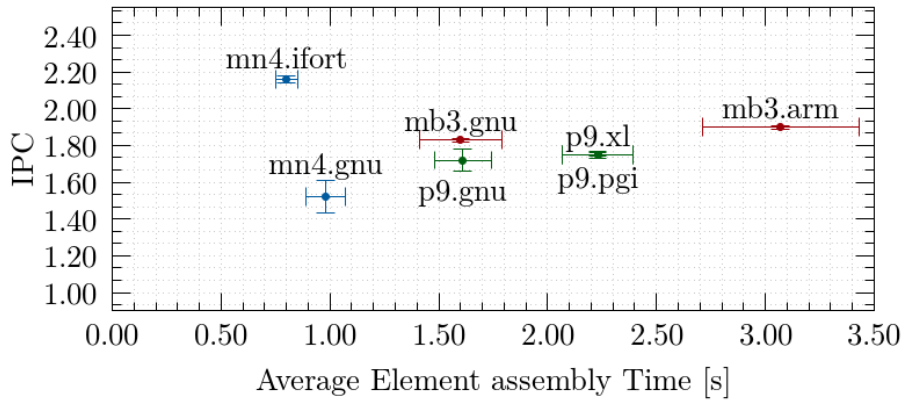

Figure 6.3: IPC and average duration of the element assembly

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 6.4 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 that, on Dibona, there is a higher variability than in the rest of the machines.

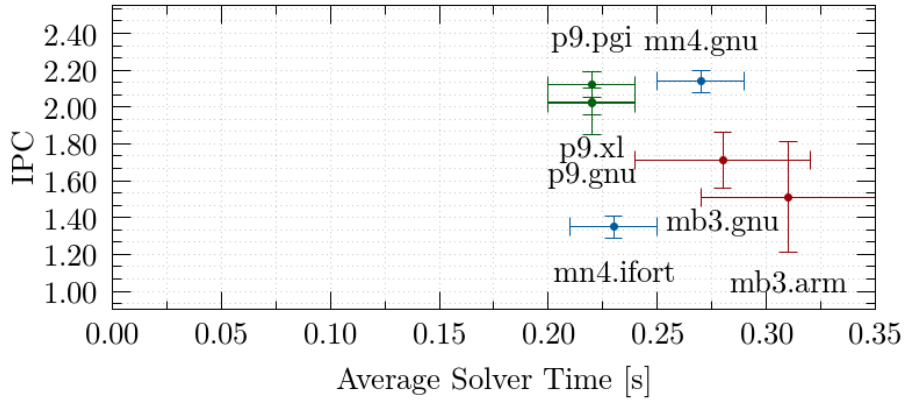

Figure 6.4: IPC and average duration of the solver

Figure 6.5 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.

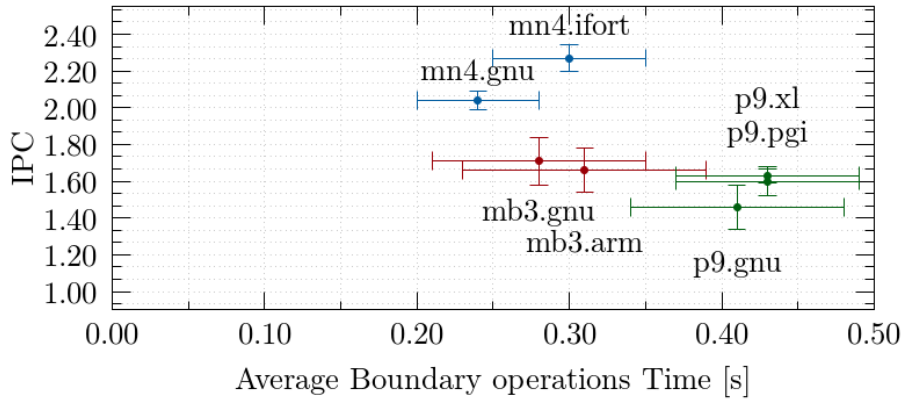

Figure 6.5: IPC and average duration of the boundary ops.

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.

Vectorization

In Figures 6.3, 6.5, and 6.4 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 6.1 shows the PAPI counters we used to measure compiler autovectorization on each machine and their corresponding description.

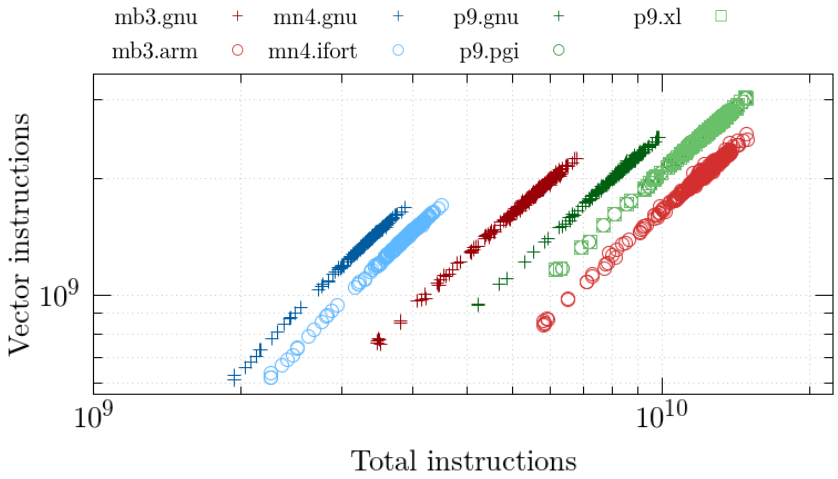

Figure 6.6: Compiler autovectorization in element assembly

Figure 6.6 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 3.5.

Our measurements show that the binaries with a lower elapsed time in Figure 6.3 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.

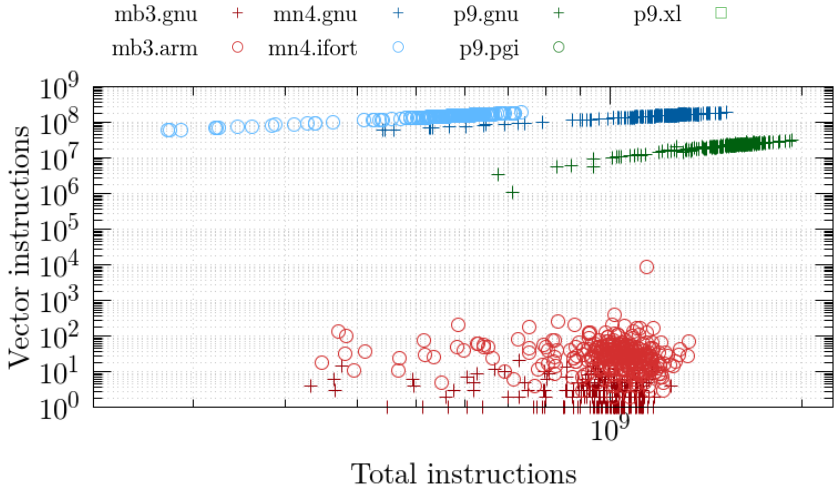

Figure 6.7: Compiler autovectorization in solver

Figure 6.7 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 6.2 appears to be memory bound, so it should run faster on a platform with higher memory bandwidth like Dibona. However, Figure 6.4 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.

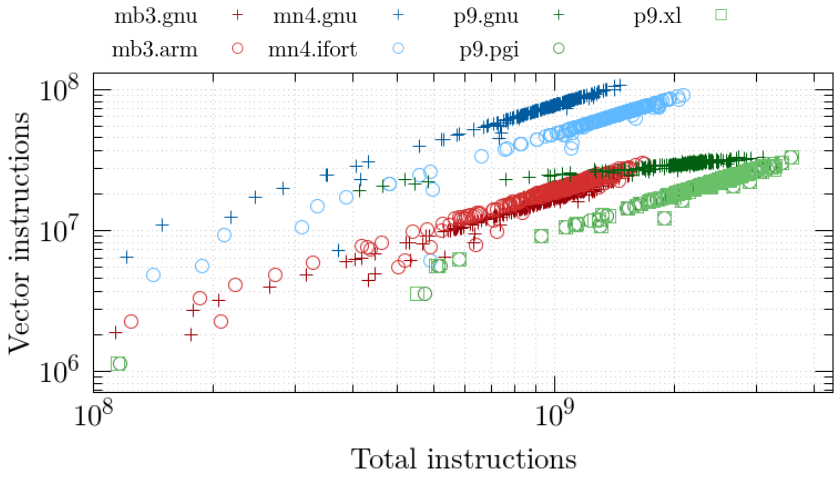

Figure 6.8: Compiler autovectorization in boundary ops.

Figure 6.8 shows the measurements in the boundary operations phase. In Figure 6.5 we showed that executions in Power9 were slower and we now show that a higher number of executed instructions may be the cause.

6.3 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.

Scalability airplane 31.5 million elements

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

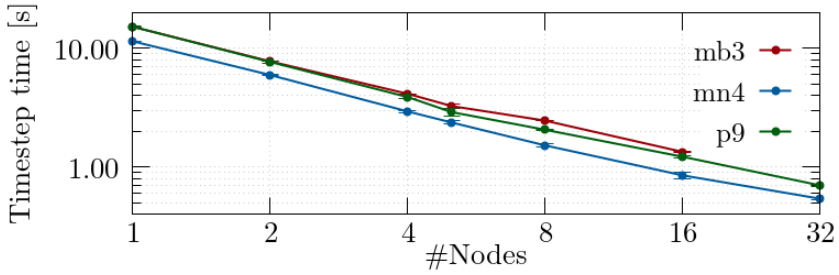

Figure 6.9: Time step scalability in all clusters

In Figure 6.9, 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.

In Figure 6.10, 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.

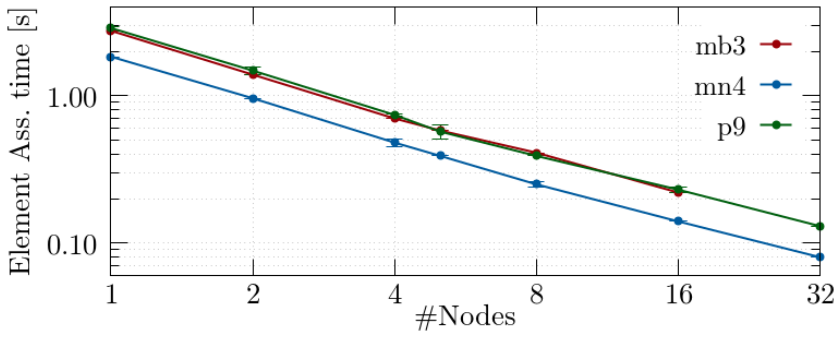

Figure 6.10: Element assembly scalability in all clusters

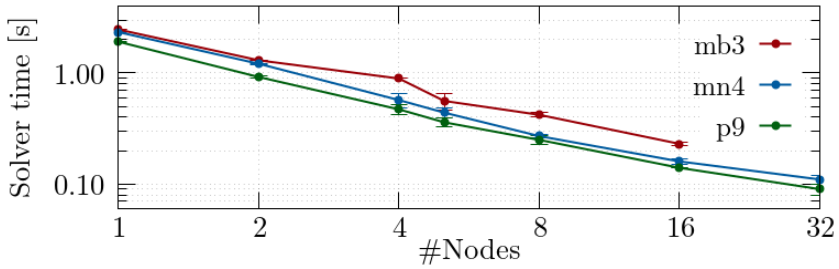

Figure 6.11: Solver scalability in all clusters

In Figure 6.11, 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 6.4, 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 6.4, we concluded that GNU is not able to generate a binary that exploits the vector units of the core.

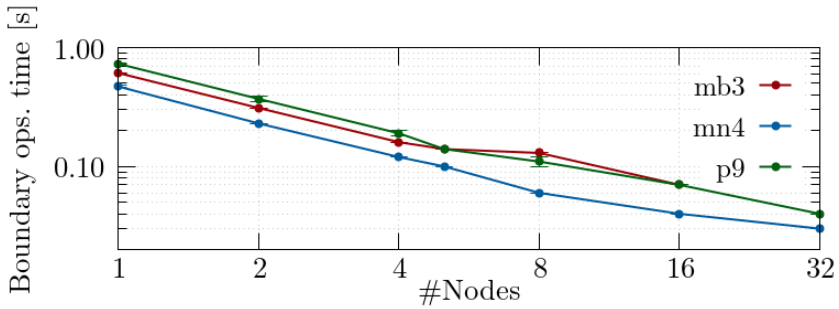

Figure 6.12: Boundary ops. scalability in all clusters

Finally, in Figure 6.12, 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 6.1, is highly dependent on the partition (and consequently on the number of MPI processes). The Load Balance 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., Figure 6.5).

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.

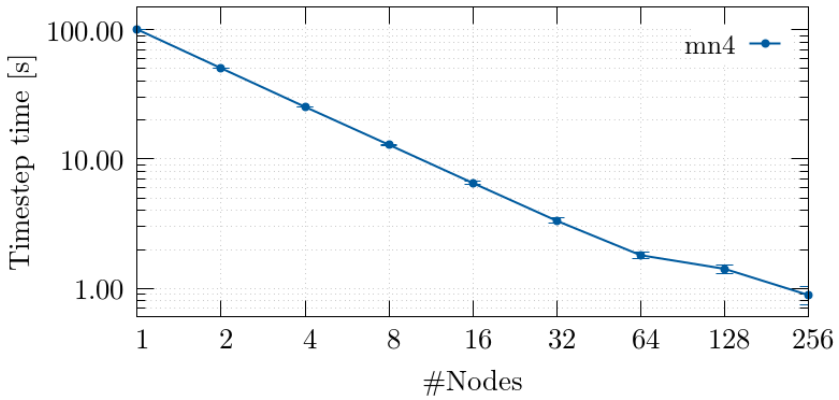

Figure 6.13: Time step scalability in MareNostrum4

In Figure 6.13, we show the scalability of a single time step when simulating the 252 million elements airplane 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).

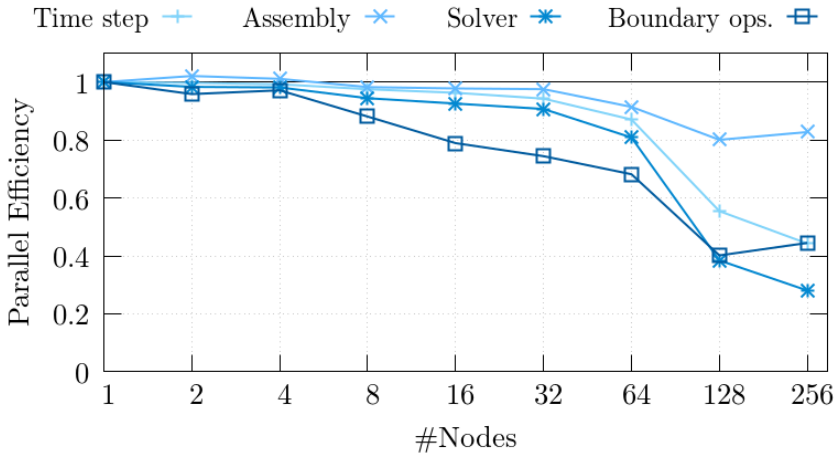

Figure 6.14: Parallel efficiency up to 256 nodes

In Figure 6.14, 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 takes 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 6.15, 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 6.14.

In Figure 6.16, 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 performance loss observed in Figure 6.14 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.

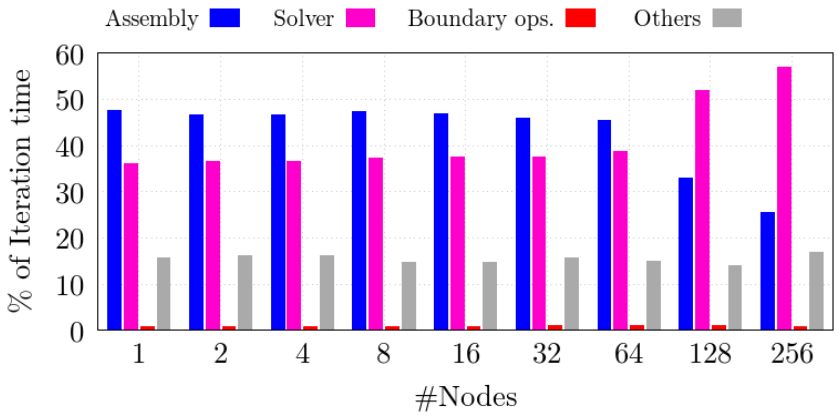

Figure 6.15: Percentage of time spent in each phase

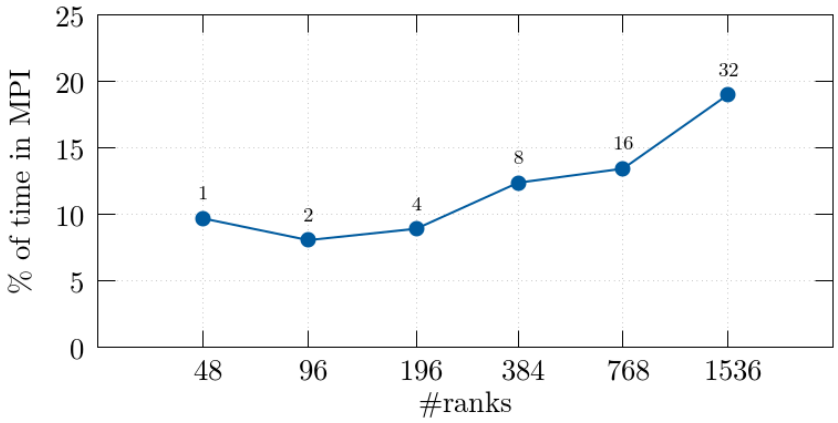

Figure 6.16: Percentage of time in MPI up to 32 nodes

6.4 Efficiency model

In Figure 6.1 we report a timeline of one iteration of Alya running in one node of MareNostrum4 using 48 MPI processes. In this section, I present a performance study of Alya running on MareNostrum4 and compiled with the Intel Compiler. The FOA includes 4 time steps of the iterative phase of the application. Appendix B contains the efficiency model tables for the three clusters that I cover in this thesis.

Single-node analysis

In Table 6.2 we report the POP efficiency metrics when running Alya on a single node of MareNostrum4. The header of the columns express the number of processes per node. The reader should remember that we run Alya with a fixed number of processes (48), spawning an increasing number of processes per node in each test. So, the column with the header "2" reports the efficiencies of Alya running with 48 MPI processes, spawning 2 processes per node on 24 compute nodes of MareNostrum4.

We see that the load balance presents a low value that remains constant for the different runs. This behaviour is as expected because all runs refers to a partition of the problem into 48 MPI ranks. The load balance problem re-appear in the multi node study and we study it in more details in the next Section where we can see how it changes when increasing the number of MPI processes (partitions of the problem).

As often happens, Table 6.2 shows that after the load balance the main limiting factor when increasing the number of processes per node is the drop of the IPC.

To study this phenomenon, we use the Clustering tool introduced in Section 3.6. We clusterize each execution trace of Alya using IPC and number of instructions. We identify three main computational clusters: Residual Assembly (RAss), Timestep Computation (TsComp), Algebraic Solver (Solver). Please note that these clusters do not correspond to the phases I introduced in Figure 6.1. The phases Element assembly and Boundaries operations have merged into a single cluster called Residual Assembly. This is because the type of operations that Alya performs in those two phases are very similar and the Clustering tool identifies them as the same execution cluster. In Figure 6.17 we highlight with different colors the clusters identified by the Clustering tool over a timeline.

	2	4	8	12	24	48
Global efficiency	83.20	82.95	81.67	81.74	79.79	75.13
├ Parallel efficiency	83.20	83.17	82.31	83.01	83.01	84.12
│ ├ Load balance	83.59	83.63	82.71	83.35	83.33	84.75
│ └ Communication eff.	99.53	99.45	99.52	99.59	99.62	99.25
│ ├ Serialization eff.	99.93	99.85	99.89	99.94	99.96	99.85
│ └ Transfer eff.	99.61	99.60	99.62	99.65	99.67	99.41
└ Computation scalability	100.00	99.74	99.23	98.47	96.12	89.32
├ IPC scalability	100.00	99.74	99.25	98.47	96.15	89.24
├ Instruction scalability	100.00	100.00	100.00	100.00	100.00	100.10
└ Frequency scalability	100.00	100.00	99.98	100.00	99.97	99.99
Speedup	1.00	1.00	0.98	0.98	0.96	0.90
Average IPC	2.14	2.14	2.13	2.11	2.06	1.91
Average frequency [GHz]	2.09	2.09	2.09	2.09	2.09	2.09

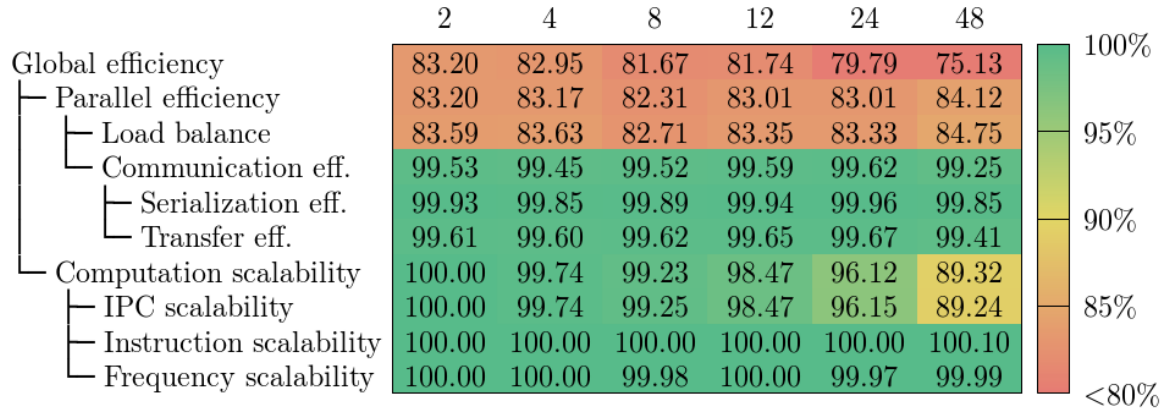

Table 6.2: Single-node efficiency metrics of Alya

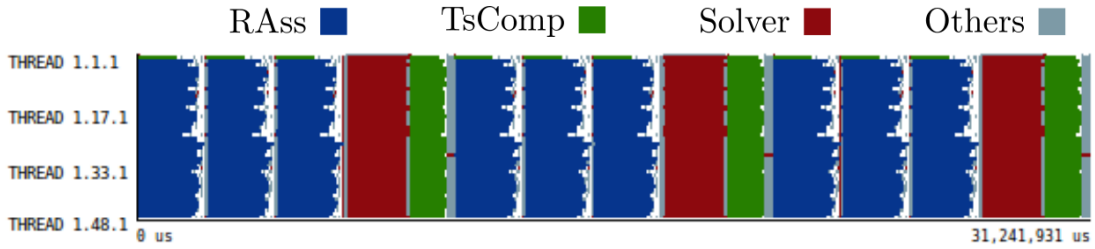

Figure 6.17: Timeline of 3 iterations highlighting the clusters

We characterize the clusters using their duration and we plot them in Figure 6.18. We notice that the Solver is the responsible for the lack of scalabilty when increasing the number of processes per node.

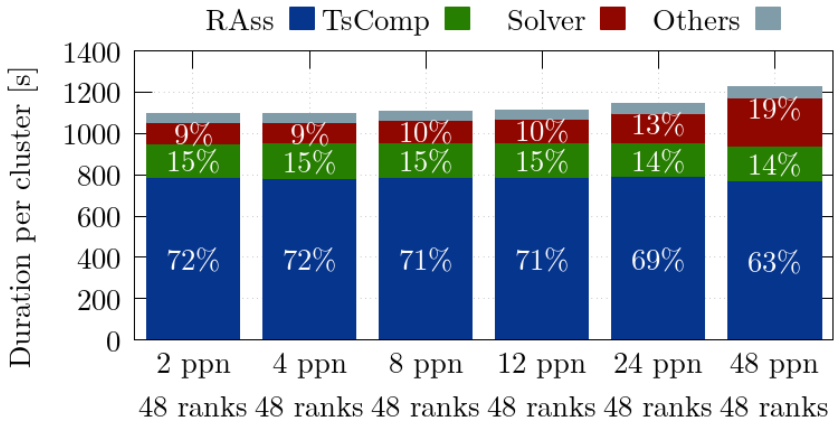

Figure 6.18: Cluster duration with 48 MPI ranks with different processes per node distribution

The IPC drop could be generated by the saturation of resources (e.g., memory bandwidth). For this reason we study the density of L3 cache misses per μsecond per socket. We perform the study on each cluster and we confirm that the Solver phase is the root cause of the IPC drop. In Figure 6.19 we report the number of L3 cache misses per μsecond per socket (y-axis) when changing the number of processes per node (x-axis). We notice that while the Residual Assembly phase and the Timestep Computation phase have a steady low L3 miss density, the Solver shows an increasing density of cache misses, corresponding to lower values of IPC when increasing the number of processes per node.

We conclude that IPC drop within the node is due to memory resource saturation when increasing the number of MPI processes per node. We suggest to find a better data layout that would allow a better cache data reuse.

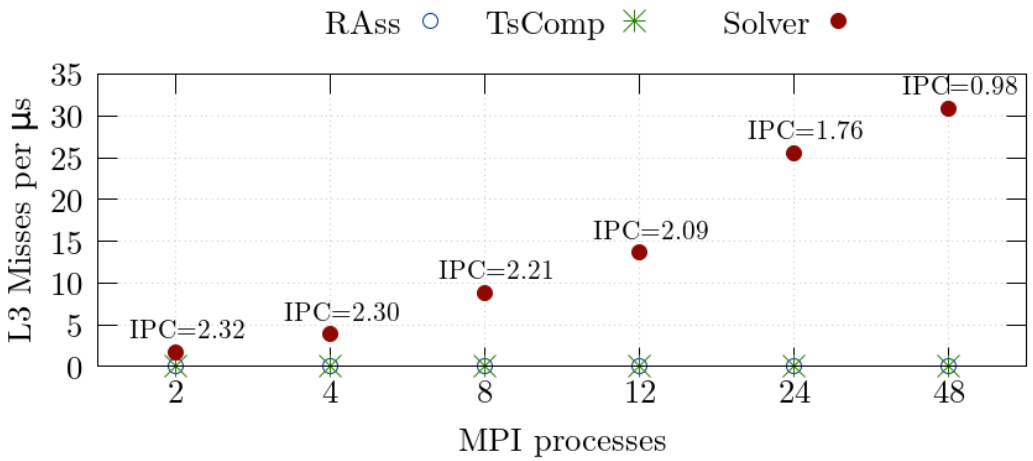

Figure 6.19: Study of the density of L3 data cache misses per μs per socket

Multi-node analysis

In Table 6.3 we can find the efficiencies obtained by Alya when using from 48 to 768 MPI processes (1 to 16 nodes). We can see that the main factor limiting scalability is the load balance.

	48	96	192	384	768
Global efficiency	83.95	80.06	81.42	76.51	65.34
├ Parallel efficiency	83.95	82.38	81.18	74.75	62.69
│ ├ Load balance	84.51	83.63	83.09	80.19	70.95
│ └ Communication eff.	99.34	98.50	97.70	93.22	88.37
│ ├ Serialization eff.	99.71	99.18	99.14	96.67	93.95
│ └ Transfer eff.	99.64	99.32	98.55	96.43	94.05
└ Computation scalability	100.00	97.18	100.30	102.35	104.22
├ IPC scalability	100.00	97.82	101.42	104.20	107.33
├ Instruction scalability	100.00	99.51	99.20	98.73	97.91
└ Frequency scalability	100.00	99.85	99.69	99.49	99.18
Speedup	1.00	1.91	3.88	7.29	12.45
Average IPC	1.91	1.87	1.93	1.99	2.05
Average frequency [GHz]	2.09	2.09	2.08	2.08	2.07

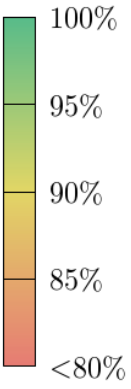

Table 6.3: Multi-node efficiency metrics of Alya

Number of processes	**48**	**96**	**192**	**384**	**768**
Load Balance	84.51%	83.63%	83.09%	80.19%	70.95%
Instruction Balance	83.10%	82.63%	82.82%	79.83%	71.03%

Table 6.4: Alya: Load and Instruction Balance when increasing number of processes

In Table 6.4 we can see the load balance in useful time as computed by the POP efficiency metrics, and the load balance in the number of instructions. We can conclude that the load balance of Alya comes from the partition of the problem, because some processes execute more instructions than others. The example considered for the present analysis is a full airplane simulation. The mesh is hybrid, composed of prisms in the boundary layer region, tetrahedra in the core flow and pyramids in the transition region. For this study, we have partitioned the mesh disregarding the type of elements, which cost for assembling the residuals is different. This relative cost is responsible for the load imbalance. This issue can be addressed using a better heuristic to partition the problem or use a dynamic load balancing mechanism as have been proved useful in other works [46, 51].

In the case of Alya we study a second factor limiting the scalability, looking at Table 6.3 we can see that after the Load Balance, the main problems are serialization and transfer. As usually serialization comes from load balancing problems in different phases with a different pattern and we have already suggested the load balance problem we study the transfer efficiency in Alya.

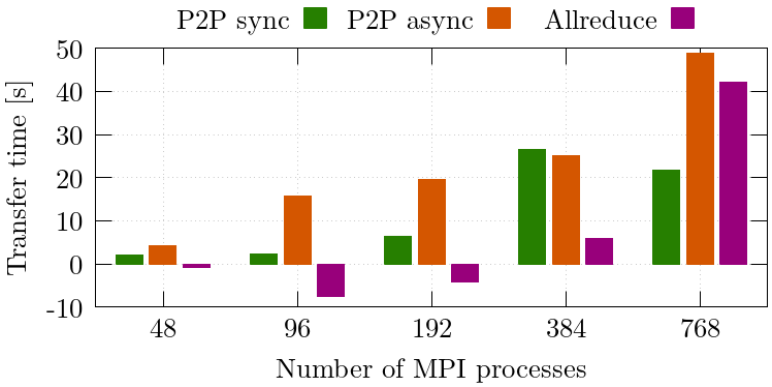

Figure 6.20: Transfer time per MPI primitive

We analyze the transfer time spent by the different class of MPI calls. The class of calls are *P2P sync* including `MPI_send`, `MPI_recv`, and `MPI_sendrecv`; *P2P async* including `MPI_isend`, `MPI_irecv`, `MPI_waitall`; *Allreduce* including `MPI_Allreduce`. In Figure 6.20 I report the time spent communicating data for different types of MPI calls. We observe that the transfer time increases steady for the asynchronous point to point MPI calls (P2P async). It also increases for the synchronous calls but stops increasing after 384 MPI processes, the amount of time spent in the AllReduce call also increases drastically for 768 MPI processes.

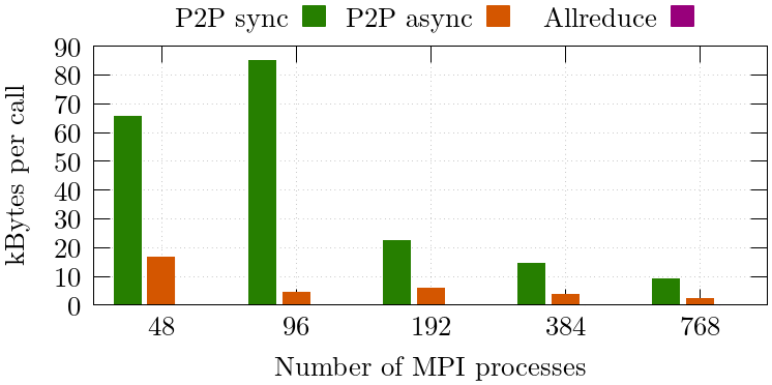

Figure 6.21: Bytes exchanged per MPI primitive

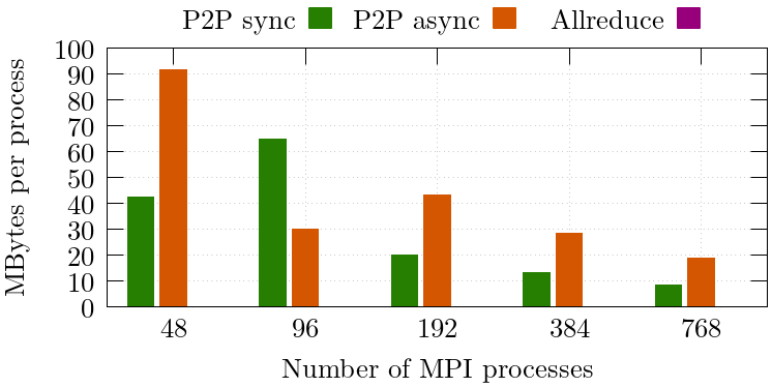

Figure 6.22: Byte exchanged by each process per MPI primitive

In Figure 6.21 and 6.22 we can see the number of bytes exchanged per MPI call and per MPI process respectively. We can see that the number of bytes exchanged per call synchronous and asynchronous MPI calls decreases drastically with the number of MPI processes. If we look at the number of bytes exchanged by process we observe that also decrease for both kinds of calls as we increase the number of MPI processes.

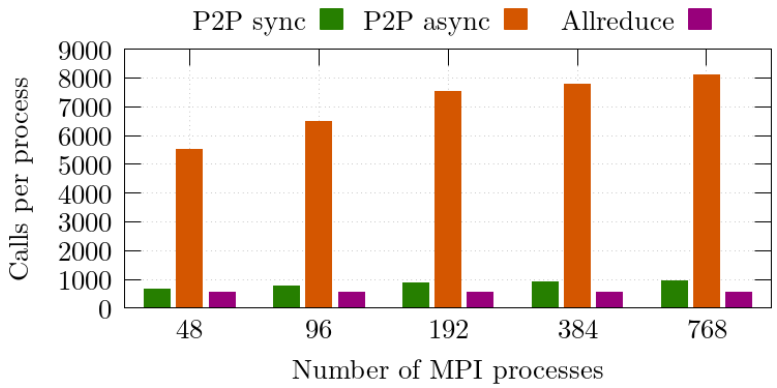

Figure 6.23: Number of calls by each process per MPI primitive

In Figure 6.23 we show the number of calls performed of each type by each process. We observe that the number of asynchronous point to point is very high and increases with the number of MPI processes. Therefore, the transfer time spent in the asynchronous point to point MPI call is due to the high number of calls and the overhead associated to them. The suggestion to address this issue is to refactor, if possible, the communications to group them and use a single call to send several data. It can happen that algorithmically this is not possible, in that case the suggestion is to look at the partition of the problem because the high number of point to point communications is a symptom of having too many neighbours.

Chapter 7

Conclusions

Ahora os toca a vosotros.

"Now is up to you."

— Agustin Fernández

In this thesis, I have presented an evaluation methodology for HPC clusters that works across multiple CPU architectures and can be applied under a short time constraint. The evaluation is done at three distinct levels: micro-benchmarks, which stress specific components of the system; system software, which evaluates the software tools that are available to the user; and a performance study of a scientific application. To bridge these three levels, I leveraged two theoretical performance models, the Roofline model and the POP Efficiency model, which I constructed with empirical measurements.

In Chapter 4, I showed the results of the first level of the evaluation model. Micro-benchmarks offer a simple approach to focus on a single element of the machine at the time. The evaluation of the floating point throughput presented in Section 4.1 demonstrates that it is easy to write a code to measure a performance close to the theoretical peak provided by the manufacturer or extrapolated from the technical specifications of the machine. On the other hand, the measurements of bandwidth and latency of the memory hierarchy in Sections 4.2, and 4.3 show that there are multiple benchmarks available. It can be difficult to determine which benchmark is more suitable for the evaluation methodology. Moreover, the complexity of the memory hierarchies in modern processors makes it difficult to design experiments with consistent results. As an example, the memory latency measurements I presented in Section 4.5 show promising results but require further investigation to be able to fully characterize the memory hierarchies of the three machines. In general, there is little information about the theoretical figures of the cache levels, which means that we cannot know how close are our measurements to the peak bandwidth or access time.

In Chapter 5, I presented an evaluation of some of the software tools in each machine. Firstly, the compiler study shows that using optimization flags yield a better performance than when compiling a binary with the default compiler optimizations. However, the flags that the system administrator provides may not leverage the micro-architecture features to the fullest (e.g., using the latest vectorization instructions in MareNostrum4). Secondly, the OSU benchmarks results show that the MPI implementation introduces multiple layers of abstraction over the raw network protocol. These layers may add communication overheads.

In Chapter 6, I conduct a performance study of a Computational Fulid Dynamics application. To this end, I leverage an efficiency model developed at BSC. This model works across multiple CPU architectures and can be applied if the machine support a basic set of PAPI counters and the application is MPI-only. The efficiency model gives a general idea to spot the eventual performance scalability bottlenecks. However, the model does not give a detailed answer and requires further investigation.

All in all, the evaluation methodology I presented in this thesis is a solid start point that build on top of the work I have contributed to since I started working at BSC. In the future, I would like to generalize further this methodology and apply it to new HPC machines. In addition, I would like to incorporate the performance evaluation of hardware accelerators and power consumption analysis.

Acronyms

ANL Argonne National Laboratory.

B/s Bytes per second.

BSC Barcelona Supercomputing Center.

CFD Computational Fulid Dynamics.

CoE Center of Excellence.

CPU Central Processing Unit.

EPI European Processor Initiative.

Flop/s Floating-point operations per second.

FOA Focus Of Analysis.

FPU Floating Point Unit.

HPC High-Performance Computing.

HPCG High-Performance Conjugate Gradient.

HPL High-Performance Linpack.

IP Intellectual Property.

IPC Instructions per Cycle.

ISA Instruction Set Architecture.

MPI Message Passing Interface.

NUMA Non Uniform Memory Access.

ORNL Oak Ridge National Laboratory.

PAPI Performance Application Programming Interface.

POP Performance Optimization and Productivity.

PSU Power Supply Unit.

SIMD Single-Instruction Multiple-Data.

SMT Simultaneous Multi-Threading.

SSD Solid State Drive.

T/s Transfers per second.

TX2 ThunderX2.

Appendix A

Reproducibility

Reproducibility is one of the most important aspects in scientific literature. Throughout this thesis, I tried to organize all the experiments and make available as much information as possible so they can be reproduced. The this thesis is hosted at a public `git` repository[1]. The repository is organized as follows:

```
hpc-systems-evaluation/
+-- experiments
|      +-- micro-benchmarks
|      +-- system-software
+-- latex
+-- plots
+-- python-virtualenv
```

The `experiments` directory contains all the source code and scripts I used to conduct the experiments I presented in this thesis. Since the version of Alya that I used is closed source, I cannot provide the source code. The `latex` directory contains the LaTeXsource of this very document. The `plots` and `python-virtualenv` directories contain data and scripts to remake the plots I presented in this document. Section A explains how to install and run the environment to reproduce the plots.

In addition to the resources found in the `git` repository, all the raw data from the experiments is available in a public Google Spreadsheet[2].

Generating plots

All plots, with exception of the figures shown in Chapter 6 and Appendix B, presented in this thesis are generated using matplotlib[3]. To remake the plots I prepared a python virtual environment with the required packages.

First, install `virtualenv`:

```
$ sudo apt install virtualenv
```

Then, create a new virtual environment. **Make sure you point to a valid python3 installation**:

```
$ virtualenv -p /usr/bin/python3 my-virtualenv
Already using interpreter /usr/bin/python3
Using base prefix '/usr'
New python executable in /tmp/my-virtualenv/bin/python3
Also creating executable in /tmp/my-virtualenv/bin/python
Installing setuptools, pkg_resources, pip, wheel...done.

$ tree -d -L 1 my-virtualenv/
my-virtualenv/
+-- bin
+-- include
+-- lib
+-- share

4 directories
```

[1] https://repo.hca.bsc.es/gitlab/fixers/hpc-systems-evaluation
[2] https://drive.google.com/drive/folders/1SqS1JQjzWGOhgnE7ndkphLIzwnHLcjJo?usp=sharing
[3] https://matplotlib.org/

To activate the virtual environment, you simply source the `activate` file inside the `bin` subdirectory:

```
$ source my-virtualenv/bin/activate
(my-virtualenv) \$
```

By default, you should have a virtual environment with no extra packages installed:

```
$ pip list
Package        Version
-------------- -------
pip            20.0.2
pkg-resources  0.0.0
setuptools     46.0.0
wheel          0.34.2
```

To install the necessary packages to remake the plots, use the `requirements.txt` file:

```
$ pip install -r requirements.txt
[...]
$ pip list
Package           Version
----------------- -------
cycler            0.10.0
kiwisolver        1.1.0
matplotlib        3.0.3
numpy             1.18.1
pandas            0.24.2
pip               20.0.2
pkg-resources     0.0.0
pyparsing         2.4.6
python-dateutil   2.8.1
pytz              2019.3
setuptools        46.0.0
six               1.14.0
wheel             0.34.2
```

You can now remake the plots by executing the python scripts under the `plots` subdirectory.
Exit the terminal to close the virtual environment.

Appendix B

Efficiency model tables

Dibona - Arm

At the time of writing, there were not enough resources in Dibona to perform the efficiency study.

MareNostrum4 - Intel

Tables B.1 and B.2 show the efficiency metrics of Alya compiled with the Intel Compiler and run in MareNostrum4. These tables are the same as the ones shown in Chapter 6.

	2	4	8	12	24	48
Global efficiency	83.20	82.95	81.67	81.74	79.79	75.13
├ Parallel efficiency	83.20	83.17	82.31	83.01	83.01	84.12
│ ├ Load balance	83.59	83.63	82.71	83.35	83.33	84.75
│ └ Communication eff.	99.53	99.45	99.52	99.59	99.62	99.25
│ ├ Serialization eff.	99.93	99.85	99.89	99.94	99.96	99.85
│ └ Transfer eff.	99.61	99.60	99.62	99.65	99.67	99.41
└ Computation scalability	100.00	99.74	99.23	98.47	96.12	89.32
├ IPC scalability	100.00	99.74	99.25	98.47	96.15	89.24
├ Instruction scalability	100.00	100.00	100.00	100.00	100.00	100.10
└ Frequency scalability	100.00	100.00	99.98	100.00	99.97	99.99
Speedup	1.00	1.00	0.98	0.98	0.96	0.90
Average IPC	2.14	2.14	2.13	2.11	2.06	1.91
Average frequency [GHz]	2.09	2.09	2.09	2.09	2.09	2.09

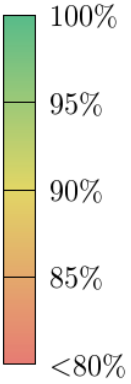

Table B.1: MareNostrum4 - Single-node efficiency metrics

Power9 - IBM

Tables B.3 and B.4 show the efficiency metrics of Alya compiled with the GNU Compiler and run in Power9.

	48	96	192	384	768
Global efficiency	83.95	80.06	81.42	76.51	65.34
├ Parallel efficiency	83.95	82.38	81.18	74.75	62.69
│ ├ Load balance	84.51	83.63	83.09	80.19	70.95
│ └ Communication eff.	99.34	98.50	97.70	93.22	88.37
│ ├ Serialization eff.	99.71	99.18	99.14	96.67	93.95
│ └ Transfer eff.	99.64	99.32	98.55	96.43	94.05
└ Computation scalability	100.00	97.18	100.30	102.35	104.22
├ IPC scalability	100.00	97.82	101.42	104.20	107.33
├ Instruction scalability	100.00	99.51	99.20	98.73	97.91
└ Frequency scalability	100.00	99.85	99.69	99.49	99.18
Speedup	1.00	1.91	3.88	7.29	12.45
Average IPC	1.91	1.87	1.93	1.99	2.05
Average frequency [GHz]	2.09	2.09	2.08	2.08	2.07

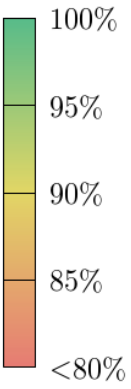

Table B.2: MareNostrum4 - Multi-node efficiency metrics

	10	16	20	40
Global efficiency	82.66	81.47	79.04	69.65
├ Parallel efficiency	82.66	81.54	81.47	83.83
│ ├ Load balance	84.13	82.58	82.24	84.08
│ └ Communication eff.	98.25	98.74	99.07	99.71
│ ├ Serialization eff.	98.44	98.96	99.25	99.89
│ └ Transfer eff.	99.80	99.77	99.81	99.82
└ Computation scalability	100.00	99.91	97.02	83.08
├ IPC scalability	100.00	99.53	98.61	97.28
├ Instruction scalability	100.00	100.27	100.25	100.64
└ Frequency scalability	100.00	100.11	98.15	84.86
Speedup	1.00	0.99	0.96	0.84
Average IPC	1.69	1.68	1.67	1.64
Average frequency [GHz]	3.76	3.76	3.69	3.19

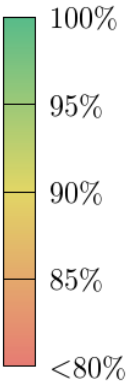

Table B.3: Power9 - Single-node efficiency metrics

	40	80	160
Global efficiency	83.83	78.90	62.93
├ Parallel efficiency	83.83	78.53	62.68
│ ├ Load balance	84.08	83.86	79.71
│ └ Communication eff.	99.71	93.64	78.64
│ ├ Serialization eff.	99.89	93.88	79.04
│ └ Transfer eff.	99.82	99.74	99.49
└ Computation scalability	100.00	100.47	100.39
├ IPC scalability	100.00	101.12	101.30
├ Instruction scalability	100.00	99.28	98.78
└ Frequency scalability	100.00	100.08	100.33
Speedup	1.00	1.88	3.00
Average IPC	1.64	1.66	1.66
Average frequency [GHz]	3.19	3.19	3.20

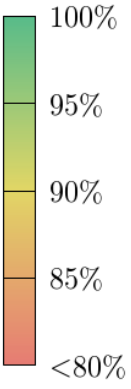

Table B.4: Power9 - Multi-node efficiency metrics

Bibliography

[1] Fabio F. Banchelli Gracia, Daniel Ruiz, Ying Hao Xu Lin, and Filippo Mantovani. Is Arm software ecosystem ready for HPC? November 2017. Accepted: 2017-11-02T11:24:26Z. 1, 2.2

[2] Fabio Banchelli et al. MB3 D6.9 – Performance analysis of applications and mini-applications and benchmarking on the project test platforms. Technical report, 2019. 1, 2.2, 3.3, 3.3, 4.5

[3] Filippo Mantovani, Marta Garcia-Gasulla, Jose Gracia, Esteban Stafford, Fabio Banchelli, Marc Josep-Fabrego, Joel Criado-Ledesma, and Mathias Nachtmann. Performance and energy consumption of HPC workloads on a cluster based on Arm ThunderX2 CPU. *Future generation computer systems*, 2020 – in press. 1, 2.2, 3.3

[4] F. Banchelli, K. Peiro, A. Querol, G. Ramirez-Gargallo, G. Ramirez-Miranda, J. Vinyals, P. Vizcaino, M. Garcia-Gasulla, and F. Mantovani. Performance study of hpc applications on an arm-based cluster using a generic efficiency model. In *2020 28th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP)*, pages 167–174, 2020. 1, 2.2, 3.6, 3.6

[5] Fabio Banchelli, Marta Garcia-Gasulla, Guillaume Houzeaux, and Filippo Mantovani. Benchmarking of state-of-the-art hpc clusters with a production cfd code. In *Proceedings of the Platform for Advanced Scientific Computing Conference*, PASC 20, New York, NY, USA, 2020. Association for Computing Machinery. 1

[6] Michael Feldman. China Fleshes Out Exascale Design for Tianhe-3 Supercomputer. https://www.nextplatform.com/2019/05/02/china-fleshes-out-exascale-design-for-tianhe-3/ - Last accessed Apr. 2020, May 2019. 2.1

[7] U.S. Department of Energy and Intel to deliver first exascale supercomputer. https://www.anl.gov/article/us-department-of-energy-and-intel-to-deliver-first-exascale-supercomputer - Last accessed Apr. 2020. 2.1

[8] U.S. Department of Energy and Cray to Deliver Record-Setting Frontier Supercomputer at ORNL. https://www.ornl.gov/news/us-department-energy-and-cray-deliver-record-setting-frontier-supercomputer-ornl - Last accessed Apr. 2020. 2.1

[9] Eurohpc. http://eurohpc.eu/ - Last accessed Apr. 2020. 2.1

[10] European processor initiative. https://www.european-processor-initiative.eu/ - Last accessed Apr. 2020. 2.1

[11] Top500, November 2018. 2.1

[12] Jack J. Dongarra, Piotr Luszczek, and Antoine Petitet. The LINPACK Benchmark: past, present and future. *Concurrency and Computation: Practice and Experience*, 15(9):803–820, 2003. _eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/cpe.728. 2.1

[13] Hpl - a portable implementation of the High-Performance linpack benchmark for Distributed-Memory computers. https://www.netlib.org/benchmark/hpl/ - Last accessed Apr. 2020. 2.1

[14] Michael A Heroux, Jack Dongarra, and Piotr Luszczek. HPCG Technical Specification. page 21. 2.1

[15] Jens Domke, Kazuaki Matsumura, Mohamed Wahib, Haoyu Zhang, Keita Yashima, Toshiki Tsuchikawa, Yohei Tsuji, Artur Podobas, and Satoshi Matsuoka. Double-precision FPUs in High-Performance Computing: an Embarrassment of Riches? *arXiv:1810.09330 [cs]*, October 2018. arXiv: 1810.09330. 2.1

[16] Green500, November 2019. 2.1

[17] Michael Feldman. Arm Supercomputer Captures The Energy Efficiency Crown, November 2019. Library Catalog: www.nextplatform.com Section: HPC. 2.1

[18] Keren Bergman, Shekhar Borkar, Dan Campbell, William Carlson, William Dally, Monty Denneau, Paul Franzon, William Harrod, Jon Hiller, Sherman Karp, Stephen Keckler, Dean Klein, Robert Lucas, Mark Richards, Al Scarpelli, Steven Scott, Allan Snavely, Thomas Sterling, R. Stanley Williams, Katherine Yelick, Keren Bergman, Shekhar Borkar, Dan Campbell, William Carlson, William Dally, Monty Denneau, Paul Franzon, William Harrod, Jon Hiller, Stephen Keckler, Dean Klein, Peter Kogge, R. Stanley Williams, and Katherine Yelick. ExaScale Computing Study: Technology Challenges in Achieving Exascale Systems Peter Kogge, Editor & Study Lead. Technical report, 2008. 2.1

[19] Stijn Heldens, Pieter Hijma, Ben Van Werkhoven, Jason Maassen, Adam S. Z. Belloum, and Rob V. Van Nieuwpoort. The Landscape of Exascale Research: A Data-Driven Literature Analysis, March 2020. 2.1, 3.1

[20] Amd epyc® 7002 series processors. - Last accessed Apr. 2020. 2.1

[21] Michael Feldman. Arms Methodical March to HPC Adoption, November 2019. Library Catalog: www.nextplatform.com Section: HPC. 2.1, 2.2

[22] Timothy Prickett Morgan. Stacking Up Arm Server Chips Against X86, March 2020. Library Catalog: www.nextplatform.com Section: Compute. 2.1

[23] Mont-blanc project. https://www.montblanc-project.eu/ - Last accessed Apr. 2020. 2.1

[24] Nikola Rajovic, Paul M Carpenter, Isaac Gelado, Nikola Puzovic, Alex Ramirez, and Mateo Valero. Supercomputing with commodity CPUs: Are mobile SoCs ready for HPC? In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, page 40. ACM, 2013. 2.1

[25] Dibona cluster - MONTBLANC-3. http://montblanc-project.eu/prototypes, 2018. 2.1

[26] Fabio Banchelli and Filippo 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)*, pages 41–50, November 2018. 2.1

[27] Yi-Chao Wang, Jin-Kun Chen, Bin-Rui Li, Si-Cheng Zuo, William Tang, Bei Wang, Qiu-Cheng Liao, Rui Xie, and James Lin. An Empirical Study of HPC Workloads on Huawei Kunpeng 916 Processor. In *2019 IEEE 25th International Conference on Parallel and Distributed Systems (ICPADS)*, pages 360–367, December 2019. ISSN: 1521-9097. 2.2

[28] Kazuhiko Komatsu, Shintaro Momose, Yoko Isobe, Osamu Watanabe, Akihiro Musa, Mitsuo Yokokawa, Toshikazu Aoyama, Masayuki Sato, and Hiroaki Kobayashi. Performance Evaluation of a Vector Supercomputer SX-Aurora TSUBASA. In *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 685–696, November 2018. 2.2

[29] Enrico Calore, Alessandro Gabbana, Sebastiano Fabio Schifano, and Raffaele Tripiccione. ThunderX2 Performance and Energy-Efficiency for HPC Workloads. *Computation*, 8(1):20, March 2020. Number: 1 Publisher: Multidisciplinary Digital Publishing Institute. 2.2, 4.4

[30] Adrian Jackson, Andrew Turner, Michle Weiland, Nick Johnson, Olly Perks, and Mark Parsons. Evaluating the Arm Ecosystem for High Performance Computing. In *Proceedings of the Platform for Advanced Scientific Computing Conference*, PASC '19, pages 1–11, Zurich, Switzerland, June 2019. Association for Computing Machinery. 2.2

[31] S D Hammond, C Hughes, M J Levenhagen, C T Vaughan, A J Younge, B Schwaller, M J Aguilar, K T Pedretti, and J H Laros. Evaluating the Marvell ThunderX2 Server Processor for HPC Workloads. page 8. 2.2

[32] Simon McIntoshSmith, James Price, Tom Deakin, and Andrei Poenaru. A performance analysis of the first generation of HPC-optimized Arm processors. *Concurrency and Computation: Practice and Experience*, 31(16):e5110, 2019. _eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/cpe.5110. 2.2

[33] Vernica G. Vergara Larrea, Wayne Joubert, Michael J. Brim, Reuben D. Budiardja, Don Maxwell, Matt Ezell, Christopher Zimmer, Swen Boehm, Wael Elwasif, Sarp Oral, Chris Fuson, Daniel Pelfrey, Oscar Hernandez, Dustin Leverman, Jesse Hanley, Mark Berrill, and Arnold Tharrington. Scaling the Summit: Deploying the Worlds Fastest Supercomputer. In Michle Weiland, Guido Juckeland, Sadaf Alam, and Heike Jagode, editors, *High Performance Computing*, Lecture Notes in Computer Science, pages 330–351, Cham, 2019. Springer International Publishing. 2.2

[34] Enrico Calore et al. Advanced performance analysis of HPC workloads on Cavium ThunderX. In *2018 International Conference on High Performance Computing & Simulation (HPCS)*, pages 375–382, 2018. 3.2

[35] Sudharshan S. Vazhkudai, Bronis R. de Supinski, Arthur S. Bland, et al. 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, pages 52:1–52:12, Piscataway, NJ, USA, 2018. IEEE Press. 3.3

[36] Samuel Williams, Andrew Waterman, and David Patterson. Roofline: An Insightful Visual Performance Model for Multicore Architectures. *Commun. ACM*, 52(4):65–76, April 2009. 3.4

[37] A. Ilic, F. Pratas, and L. Sousa. Cache-aware Roofline model: Upgrading the loft. *IEEE Computer Architecture Letters*, 13(1):21–24, January 2014. 3.4

[38] D. Marques, H. Duarte, A. Ilic, L. Sousa, R. Belenov, P. Thierry, and Z. A. Matveev. Performance Analysis with Cache-Aware Roofline Model in Intel Advisor. In *2017 International Conference on High Performance Computing Simulation (HPCS)*, pages 898–907, July 2017. 3.4

[39] Michael Wagner, Stephan Mohr, Judit Giménez, and Jesús Labarta. A structured approach to performance analysis. In *International Workshop on Parallel Tools for High Performance Computing*, pages 1–15. Springer, 2017. 3.5

[40] Vincent Pillet, Vincent Pillet, Jess Labarta, Toni Cortes, Toni Cortes, Sergi Girona, Sergi Girona, and Departament D'arquitectura De Computadors. PARAVER: A Tool to Visualize and Analyze Parallel Code. Technical report, In WoTUG-18, 1995. 3.6

[41] Sergi Girona, Jess Labarta, and Rosa M. Badia. Validation of Dimemas Communication Model for MPI Collective Operations. In Jack Dongarra, Peter Kacsuk, and Norbert Podhorszki, editors, *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, Lecture Notes in Computer Science, pages 39–46. Springer Berlin Heidelberg, 2000. 3.6

[42] Larry W McVoy, Carl Staelin, et al. lmbench: Portable tools for performance analysis. In *USENIX annual technical conference*, pages 279–294. San Diego, CA, USA, 1996. 4.3

[43] Yu Jung Lo, Samuel Williams, Brian Van Straalen, Terry J. Ligocki, Matthew J. Cordery, Nicholas J. Wright, Mary W. Hall, and Leonid Oliker. Roofline Model Toolkit: A Practical Tool for Architectural and Program Analysis. In Stephen A. Jarvis, Steven A. Wright, and Simon D. Hammond, editors, *High Performance Computing Systems. Performance Modeling, Benchmarking, and Simulation*, Lecture Notes in Computer Science, pages 129–148, Cham, 2015. Springer International Publishing. 4.4

[44] Aparna Sasidharan and Marc Snir. Miniamr-a miniapp for adaptive mesh refinement. Technical report, 2016. 5.3

[45] Jiuxing Liu, Balasubramanian Chandrasekaran, Jiesheng Wu, Weihang Jiang, Sushmitha Kini, Weikuan Yu, Darius Buntinas, Pete Wyckoff, and Dhabaleswar K Panda. Performance comparison of mpi implementations over infiniband, myrinet and quadrics. In *SC'03: Proceedings of the 2003 ACM/IEEE Conference on Supercomputing*, pages 58–58. IEEE, 2003. 5.4

[46] Marta Garcia-Gasulla, Filippo Mantovani, Marc Josep-Fabrego, Beatriz Eguzkitza, and Guillaume Houzeaux. Runtime mechanisms to survive new hpc architectures: A use case in human respiratory simulations. *The International Journal of High Performance Computing Applications*, 2019. 6.1, 6.4

[47] Rainald Löhner. *Applied Computational Fluid Dynamics Techniques: An Introduction Based on Finite Element Methods*. John Wiley & Sons, 2008. 6.1

[48] Rainald Löhner and Joseph D. Baum. On maximum achievable speeds for field solvers. *International Journal of Numerical Methods for Heat & Fluid Flow*, 24(7):1537–1544, 2014. 6.1

[49] Karypis G. and Kumar V. MeTis: Unstructured Graph Partitioning and Sparse Matrix Ordering System. http://www.cs.umn.edu/ metis, University of Minnesota, Minneapolis, MN, 2009. 6.1

[50] Mariano Vázquez, Guillaume Houzeaux, Seid Koric, et al. Alya: Multiphysics Engineering Simulation Toward Exascale. *Journal of Computational Science*, 14:15–27, 2016. 6.1

[51] Marta Garcia, Jesus Labarta, and Julita Corbalan. Hints to improve automatic load balancing with lewi for hybrid applications. *Journal of Parallel and Distributed Computing*, 74(9):2781 – 2794, 2014. 6.4