UNIONE EUROPEA
Fondo Sociale Europeo

PON RICERCA E INNOVAZIONE REACT EU

UNIVERSITÀ DEGLI STUDI DI TRIESTE

# UNIVERSITÀ DEGLI STUDI DI TRIESTE

## XXXVII CICLO DEL DOTTORATO DI RICERCA IN FISICA

# GREEN HPC AND BIG-DATA ANALYSIS WITH APPLICATIONS IN NUMERICAL COSMOLOGY

Settore scientifico-disciplinare: **FIS/05 ASTRONOMIA E ASTROFISICA**


DOTTORANDO / A
**GIOVANNI LACOPO**

COORDINATORE
**PROF. FRANCESCO LONGO**

SUPERVISORE DI TESI
**PROF. STEFANO BORGANI**

CO-SUPERVISORE DI TESI
**DOTT. LUCA TORNATORE**

CO-SUPERVISORE DI TESI
**DOTT. GIULIANO TAFFONI**


**ANNO ACCADEMICO 2023/2024**

# Acknowledgements

*This page intentionally left blank.*

# Abstract

High Performance Computing systems are currently becoming hugely energy consuming (Kurp, 2008), having reached several megawatts of power requested for pre-exascale and exascale machines, able to achieve a computing power of $10^{18}$ floating-point operations per second.

From the other side, increasingly complex and memory demanding simulations and data processing are nowadays needed in every field of physics, from condensed-matter physics to nuclear physics, from particle physics to astrophysics and cosmology. Focusing on the latter, current and near-future surveys spanning ranges from radio to X-rays are producing an outstanding amount of data, which need to be processed as much as possible in real-time (Mickaelian, 2016), otherwise storage problems will raise and many data seriously risk to be thrown away. Furthermore, these observations have to be compared to simulations, which are currently run with N-body (Springel, 2005) and approximated algorithms (Monaco et al., 2002), with the purpose to reconstruct the formation of cosmic structures in the universe. Largest simulations are run with trillions of particles, and naturally need supercomputers with petabytes of memory.

In order to come to an agreement between energy consumed, memory requirements and performances of both data processing and simulations, much effort has been put on the development of modern and energy efficient hardwares, like ARM CPUs, GPUs and FPGAs, and from the software side, to new programming languages and green algorithms to catch up with the technological progress. In fact, most codes require to be redesigned since otherwise they would run very inefficiently on the newest HPC machines, in other words, they need to be "parallelized".

In the first part of the thesis, I will focus on a problem raised in a code for radio imaging and due to the Message Passing Interface parallelization scheme (Clarke et al., 1994), which takes up to $\sim 90\%$ of code's runtime and this percentage is even doomed to increase by using more and more computing resources. In parallel computing this is an example of **communication-bound** problem, where parallel workers cannot work independently and they need to exchange data at different steps. To solve this issue we implemented a customized communication function which is up to 6 times faster and 7 times less energy consuming than the standard OpenMPI implementation[1]. To measure the energy difference, much effort has been done to develop a driver and a

[1] `https://www.open-mpi.org/`

software in order to read properly the hardware energy counters, which are different for each vendor (Schöne et al., 2021). The most astonishingly important result that we found is the **algorithmic imprint** in energy-to-solution, which, in a nutshell, reflects the possibility to reduce the energy consumption without impacting on the runtime. Energy is **not** solely an integral over time.

The customized communication function has then been implemented in our radio imaging code, named **RICK** (Radio Imaging Code Kernels) (De Rubeis et al., 2024). The communication algorithm impacts in the CPU version of the code, but however much effort has been put in the full GPU code version. Aside from the communication, the code performs a convolution kernel to assign point-like data to a two-dimensional grid (i.e. gridding), several Fourier Transforms and eventually the Earth curvature correction, which is especially required for large field of view (FoV) observations.

The work of the second part of the thesis has been to offload to GPU all the four code steps, and the tests against the preceding CPU version. The code has been run at the Leonardo supercomputer, available at CINECA[2], ranked as 7-th in the June 2024 Top500 list[3]. I've been involved in the GPU offloading of the gridding step, and in the implementation of GPU-GPU communication and Fourier Transform. Communication has been addressed thanks to NVIDIA Collective Communication Library[4], which utilizes an MPI-like approach, while the Fourier Transform has been addressed with cuda Fast Fourier Transform for Multi-Processing[5], which indeed exploits the Fast Fourier Transform (FFT) algorithm (Nussbaumer et al., 1982), allowing to scale with the dataset $D$ as $DlogD$ instead of $D^2$. When the problem size is large, we found that NCCL communication is 175 times faster than pure CPU communication with MPI, and cuFFTMP is around 40 times faster than CPU one, leading to an overall performance gain factor of $\sim 150$. This result is of utmost importance for the heavy datasets coming from LOFAR (van Haarlem et al., 2013), and then especially for SKA[6], which is supposed to generate hundreds of petabytes of data per year. To handle such a huge amount of data and perform almost real-time processing the full GPU version of RICK becomes a milestone, given the performance advantage.

In the last part of the thesis, I put my effort in finding the best compromise between energy-to-solution and time-to-solution for CPU and GPU configurations of RICK. Results have been achieved during the period I spent in Perth (WA), in a collaboration with the Pawsey Supercomputing Centre. My collaborators in Australia let me use their machine, Setonix[7], ranked as 28-th in the June 2024 Top500 list. The machine has been thought to be green at once, since it's partially fed with solar panels and it's cooled with the aquifer's water under Perth. Furthermore, many energy counters are available on the machine and there are user-friendly softwares which allow to access

[2] https://leonardo-supercomputer.cineca.eu/it/home-it/
[3] https://top500.org/lists/top500/list/2024/06/
[4] https://developer.nvidia.com/nccl
[5] https://docs.nvidia.com/hpc-sdk/cufftmp/index.html
[6] https://www.skatelescope.org/
[7] https://pawsey.org.au/systems/setonix/

them. In agreement with tests run on Leonardo, we found that for large datasets GPUs are much faster than CPUs, with an average factor of $11$. This is an order of magnitude smaller than the $150$ factor achieved on Leonardo, but it's explained at first since the entire dataset is smaller by a factor of $6.6$, which reduces the GPU-to-CPU computing power ratio, and secondly since the FFT algorithm is not available for non-NVIDIA GPU so far, and Setonix is a full AMD machine. In terms of energy-to-solution, GPUs are $5-6$ times more efficient on average, which is a factor of two smaller than runtime gain. However, this is explained since GPUs have a higher TDP ratio, but they're so faster than CPUs that compensate these power requirements and eventually become greener. For the sake of completeness we found that, for small datasets, GPUs turn out not to be the greenest solution, in agreement with the previous sentence. Eventually, I had the chance to measure energy consumption by fine-tuning the CPU frequency, finding that in a communication-bound code like RICK it's possible to pass from $2.6GHz$ to $1.5GHz$ with a performance drop of $8\%$ and an energy saving of $30\%$.

*This page intentionally left blank.*

# Contents

*This page intentionally left blank.*

# List of Figures

*This page intentionally left blank.*

# List of Tables

*This page intentionally left blank.*

*This page intentionally left blank.*

*This page intentionally left blank.*

*This page intentionally left blank.*

# 1 Introduction

## 1.1 HPC and its role in science

**Figure 1.1:** *Reconstructed scheme of a standard HPC centre appears. Source: https://www.trentonsystems.com/en-us/resource-hub/blog/high-performance-computing.*

The first question we would like to address is: what is High Performance Computing? There are several books explaining this new field in computer science, for instance (Aversa et al., 2004; Eijkhout et al., 2016), which address this tricky question. HPC is a co-design of hardware and software:

- **Hardware** At this level, people are working to develop modern technologies to construct always "bigger" machines, with the goal to increase their computing capabilities, universally measured in $Flops$, i.e. number of floating-point operations per second.

- **Software** From this point of view, people work to develop updated codes and algorithms whose purpose is to run efficiently in the modern hardware technologies. This raises the co-design in which people developing hardware and people developing software collaborate in an entangled fashion to allow the machines to reach their theoretical peak performance, i.e. the maximum number of $Flops$ guaranteed by the vendor.

Aside from performance, one other important aspect in which both hardware and software developers are focusing is the portability, i.e. the ability to reach a unified programming scheme which allow software engineers to write codes able to run onto as many architectures as possible. Figure 1.1 is a simplified scheme of the structure of an HPC platform. There are many servers filled with tens of **computing nodes**, i.e. boards constituted by one or more CPUs, memory banks, internal fast interconnects, like QuickPath Interconnect or UltraPath Interconnect for Intel architectures, accelerators if they are available, one or more network boards and cables, and an interface between the CPU and the accelerators/network cables, called PCIe (Peripheral Component Interconnect Express) (Wilen et al., 2003; Budruk et al., 2004; Fountain et al., 2005). Interconnects, interfaces and networks are characterized by their **bandwidth**, measured in $GB/s$, i.e. how many billions of bytes can be transferred per second.

There isn't currently any Physics field in which High Performance Computing is not necessary, due to the huge number of operations needed, which cannot be performed by hand anymore. Molecular dynamics, condensed matter physics, nuclear physics, plasma physics, particle physics and, most important for this thesis project, astrophysics and cosmology require now the calculation of interations among up to trillions of particles. Considering an average of $300B$ per particle, a straightforward calculation means that $\sim 300TB$ of memory will be needed to allocate particles' memory only. If other datasets are to be included, like for instance grids in two or three-dimensional space, the memory requirement easily reaches the $PB$ scale. It is straightforward to grasp that any single computer cannot reach such a huge amount of memory. This led to the idea of connecting together many single computers, in order to reach the requested memory by summing up all the memories of the single calculators. As already mentioned, these single computers are called computing nodes.

This forced software developers to write programs able to exploit "parallelism", i.e. the ability to use multiple computing resources at the same time by splitting the workload among the "parallel workers" (Akl, 1997).

Figure 1.2 shows schematically how parallelism works. On the right side, we see the serial computing, where different tasks with all their instructions are assigned to a single worker, which is in charge to perform all of them sequentially. On the left, the associated parallel computing scheme is displayed, in which four tasks are assigned to four different workers, which are able to execute the tasks instructions concurrently. In principle, if $N$ is the number of parallel workers, the code's workload can be split into $N$ tasks to be assigned to each worker, and if original execution time is $T$, the expected

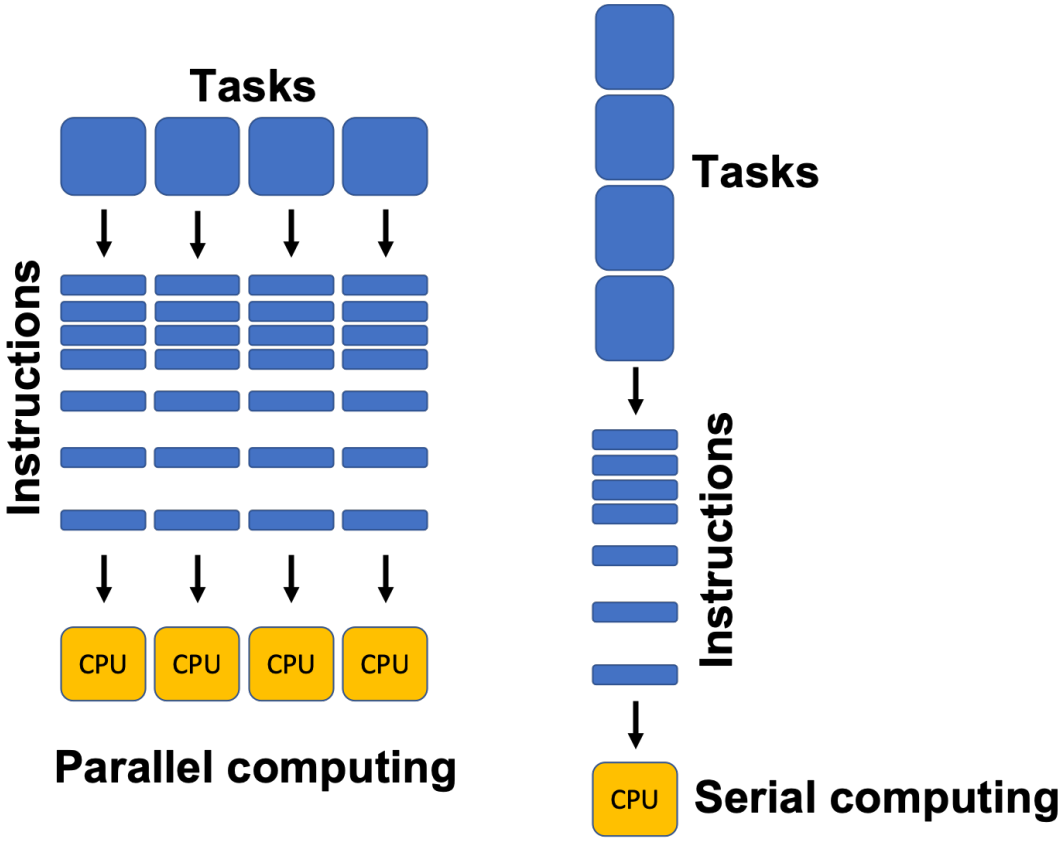



**Figure 1.2:** *Scheme of how tasks are assigned to the CPU in the serial case (on the right), and in the parallel case (on the left). Parallelism allows the CPU to perform multi-tasks and their instructions at the same time. Source:* `https://pythonnumericalmethods.studentorg.berkeley.edu/notebooks/chapter13.01-Parallel-Computing-Basics.html`.

one in the parallel case is $T/N$. Deviations with respect to this theoretical expectation will be explored in Section 1.6 and Section 1.7.

There are at least two levels of parallelism in an HPC machine:

- The parallelism thanks to different computing nodes connected with a network.
- The parallelism inside each computing node.

We will go into the detail of this distinction in the next sections. The current most powerful machines, which harness both level of parallelism, reach the order of $N \sim 10^5$ parallel workers, distributed among $\sim 10^3 - 10^4$ computing nodes. All of them use the Linux OS environment.

## 1.2 The 500 rankings


**Figure 1.3:** *Top500 logo. Source:* `https://www.pngegg.com/en/png-tfqro`.

In June 1993, HPC machines started to be classified in the Top500[1] list (see Figure 1.3), ranked by their HPL performance in $Flops$. High Performance Linpack is a benchmark (J. J. Dongarra, 1988; J. J. Dongarra et al., 2003), which measures how quickly a computer solves a dense system of $n$ linear equations, for instance:

$$\begin{cases} a_1 \times y_1 = b_1 \\ a_2 \times y_2 = b_2 \\ \vdots \quad \vdots \quad \vdots \\ a_n \times y_n = b_n \end{cases}$$

Figure 1.4 shows the first 10 machines ranked according to the Top500, updated at June 2024. There are two machines which have reached the exascale, i.e. $10^{18} Flops$, Frontier[2] and Aurora[3]. However, the achieved performance, in the fourth column, is much smaller than the theoretical peak performance, in the fifth column. In the case of Aurora, it turns out that the supercomputer reaches around $50\%$ of the theoretical performance, given as the summation of the peak performance of all its components. From the fifth to the ninth place, there are four European machines, in which there is also the Italian Leonardo supercomputer, used for many tests in this thesis. It is interesting that 9 out of 10 are equipped with **accelerators**, while the Fugaku supercomputer in Japan does not have such hardware technology. Above $1 EFlops$, the difference between the actual performance and the theoretical performance becomes dramatic.

Dense linear equation systems are extremely useful, but in computer science and especially in Physics the codes involved need to perform much more complex calculations, i.e. particle-particle interations, integrations of 2D-3D differential equations, computations of Fourier Transforms, operations on trees, and so on. In such cases, the performance obtained with HPL cannot be meaningful to shade light on the actual computing power of the machines. Furthermore, as will be discussed in Section 1.7, in these more complex codes tasks distributed among parallel workers are not indepen-

[1] `https://top500.org/`
[2] `https://www.hpe.com/emea_europe/en/compute/hpc/cray/oak-ridge-national-laboratory.html`
[3] `https://www.anl.gov/aurora`

dent, and some kind of information exchange between the workers is unavoidable. For this reason, another benchmark was developed to complement the LINPACK benchmarks, i.e. HPCG, High Performance Conjugate Gradients[4] (Heroux et al., 2013; J. Dongarra et al., 2013; Marjanović et al., 2015), which is intended to model the data access patterns of real-world applications such as sparse matrix calculations, thus testing the limitations due to memory impact and the internal interconnect of the supercomputer on its computing performance. Information exchange is thus needed and the impact of memory transfer does not allow the machine to focus only on the computations, because of **communications**, leading to the performance drop shown in the last column of Figure 1.5.

It is easy to notice that the rankings are changed with respect to the Top500 list, at least for the first $10$ machines. Now the non-accelerated Japanese Fugaku is the most powerful supercomputer in HPCG, with a peak performance of $16PFlops$. By comparing the performances in HPL and HPCG we get a factor of $28$ in degradation for Fugaku, which becomes a much more dramatic factor of $86$ and $181$ for the "exascale" machines Frontier and Aurora, respectively.

HPCG performance drop hints that exa-scale has not actually been reached since a more accurate memory impacting benchmark reveals that the nominal performance achievement is $14 - 16PFlops$, meaning that current HPC has arrived at peta-scale instead of exa-scale. It is interesting to note that the machine which is the fourth in Top500 is the first in the HPCG500 and, furthermore, is also the only machine without accelerators.

[4] `https://top500.org/lists/hpcg/2024/06/`

| Rank | System | Cores | Rmax (PFlop/s) | Rpeak (PFlop/s) | Power (kW) |
|---|---|---|---|---|---|
| 1 | **Frontier** - HPE Cray EX235a, AMD Optimized 3rd Generation EPYC 64C 2GHz, AMD Instinct MI250X, Slingshot-11, HPE<br>DOE/SC/Oak Ridge National Laboratory<br>United States | 8,699,904 | 1,206.00 | 1,714.81 | 22,786 |
| 2 | **Aurora** - HPE Cray EX - Intel Exascale Compute Blade, Xeon CPU Max 9470 52C 2.4GHz, Intel Data Center GPU Max, Slingshot-11, Intel<br>DOE/SC/Argonne National Laboratory<br>United States | 9,264,128 | 1,012.00 | 1,980.01 | 38,698 |
| 3 | **Eagle** - Microsoft NDv5, Xeon Platinum 8480C 48C 2GHz, NVIDIA H100, NVIDIA Infiniband NDR, Microsoft Azure<br>Microsoft Azure<br>United States | 2,073,600 | 561.20 | 846.84 | |
| 4 | **Supercomputer Fugaku** - Supercomputer Fugaku, A64FX 48C 2.2GHz, Tofu interconnect D, Fujitsu<br>RIKEN Center for Computational Science<br>Japan | 7,630,848 | 442.01 | 537.21 | 29,899 |
| 5 | **LUMI** - HPE Cray EX235a, AMD Optimized 3rd Generation EPYC 64C 2GHz, AMD Instinct MI250X, Slingshot-11, HPE<br>EuroHPC/CSC<br>Finland | 2,752,704 | 379.70 | 531.51 | 7,107 |
| 6 | **Alps** - HPE Cray EX254n, NVIDIA Grace 72C 3.1GHz, NVIDIA GH200 Superchip, Slingshot-11, HPE<br>Swiss National Supercomputing Centre (CSCS)<br>Switzerland | 1,305,600 | 270.00 | 353.75 | 5,194 |
| 7 | **Leonardo** - BullSequana XH2000, Xeon Platinum 8358 32C 2.6GHz, NVIDIA A100 SXM4 64 GB, Quad-rail NVIDIA HDR100 Infiniband, EVIDEN<br>EuroHPC/CINECA<br>Italy | 1,824,768 | 241.20 | 306.31 | 7,494 |
| 8 | **MareNostrum 5 ACC** - BullSequana XH3000, Xeon Platinum 8460Y+ 32C 2.3GHz, NVIDIA H100 64GB, Infiniband NDR, EVIDEN<br>EuroHPC/BSC<br>Spain | 663,040 | 175.30 | 249.44 | 4,159 |
| 9 | **Summit** - IBM Power System AC922, IBM POWER9 22C 3.07GHz, NVIDIA Volta GV100, Dual-rail Mellanox EDR Infiniband, IBM<br>DOE/SC/Oak Ridge National Laboratory<br>United States | 2,414,592 | 148.60 | 200.79 | 10,096 |
| 10 | **Eos NVIDIA DGX SuperPOD** - NVIDIA DGX H100, Xeon Platinum 8480C 56C 3.8GHz, NVIDIA H100, Infiniband NDR400, Nvidia<br>NVIDIA Corporation<br>United States | 485,888 | 121.40 | 188.65 | |

**Figure 1.4:** *First* 10 *machines in the Top500 list. Source:* `https://top500.org/lists/top500/2024/06/`.

| Rank | TOP500 Rank | System | Cores | Rmax (PFlop/s) | HPCG (TFlop/s) |
|---|---|---|---|---|---|
| 1 | 4 | **Supercomputer Fugaku** - Supercomputer Fugaku, A64FX 48C 2.2GHz, Tofu interconnect D, Fujitsu<br>RIKEN Center for Computational Science<br>Japan | 7,630,848 | 442.01 | 16004.50 |
| 2 | 1 | **Frontier** - HPE Cray EX235a, AMD Optimized 3rd Generation EPYC 64C 2GHz, AMD Instinct MI250X, Slingshot-11, HPE<br>DOE/SC/Oak Ridge National Laboratory<br>United States | 8,699,904 | 1,206.00 | 14054.00 |
| 3 | 2 | **Aurora** - HPE Cray EX - Intel Exascale Compute Blade, Xeon CPU Max 9470 52C 2.4GHz, Intel Data Center GPU Max, Slingshot-11, Intel<br>DOE/SC/Argonne National Laboratory<br>United States | 9,264,128 | 1,012.00 | 5612.60 |
| 4 | 5 | **LUMI** - HPE Cray EX235a, AMD Optimized 3rd Generation EPYC 64C 2GHz, AMD Instinct MI250X, Slingshot-11, HPE<br>EuroHPC/CSC<br>Finland | 2,752,704 | 379.70 | 4586.95 |
| 5 | 6 | **Alps** - HPE Cray EX254n, NVIDIA Grace 72C 3.1GHz, NVIDIA GH200 Superchip, Slingshot-11, HPE<br>Swiss National Supercomputing Centre (CSCS)<br>Switzerland | 1,305,600 | 270.00 | 3671.32 |
| 6 | 7 | **Leonardo** - BullSequana XH2000, Xeon Platinum 8358 32C 2.6GHz, NVIDIA A100 SXM4 64 GB, Quad-rail NVIDIA HDR100 Infiniband, EVIDEN<br>EuroHPC/CINECA<br>Italy | 1,824,768 | 241.20 | 3113.94 |
| 7 | 9 | **Summit** - IBM Power System AC922, IBM POWER9 22C 3.07GHz, NVIDIA Volta GV100, Dual-rail Mellanox EDR Infiniband, IBM<br>DOE/SC/Oak Ridge National Laboratory<br>United States | 2,414,592 | 148.60 | 2925.75 |
| 8 | 14 | **Perlmutter** - HPE Cray EX 235n, AMD EPYC 7763 64C 2.45GHz, NVIDIA A100 SXM4 40 GB, Slingshot-11, HPE<br>DOE/SC/LBNL/NERSC<br>United States | 888,832 | 79.23 | 1905.00 |
| 9 | 12 | **Sierra** - IBM Power System AC922, IBM POWER9 22C 3.1GHz, NVIDIA Volta GV100, Dual-rail Mellanox EDR Infiniband, IBM / NVIDIA / Mellanox<br>DOE/NNSA/LLNL<br>United States | 1,572,480 | 94.64 | 1795.67 |
| 10 | 15 | **Selene** - NVIDIA DGX A100, AMD EPYC 7742 64C 2.25GHz, NVIDIA A100, Mellanox HDR Infiniband, Nvidia<br>NVIDIA Corporation<br>United States | 555,520 | 63.46 | 1622.51 |

**Figure 1.5:** *First* 10 *machines in the HPCG500 list. Source:* `https://top500.org/lists/hpcg/2024/06/`.

## 1.3 The Green500 list and the green computing

In 1990's the American DOE (Department of Energy) stated that consumption due to HPC platforms was soon becoming unsustainable, urging the research centres to develop new hardware architectures in order to reduce energy consumption. This topic has been discussed in many works, we cite (Lo et al., 2010; Gai et al., 2016; More et al., 2017; Paul et al., 2023) for an introduction to the problem.

The problem easily became of utmost importance in HPC and machines started to be ranked in the Green500[5] list (Ge et al., 2007; W.-c. Feng et al., 2007; W.-C. Feng et al., 2009; W.-c. Feng et al., 2010; Scogland et al., 2011; Scogland et al., 2013), with their energy efficiency measured by $GFlops/W$, i.e. billions of floating-point operations per second per watt.

Figure 1.6 displays the first 10 machines in the Green500 list, and in the second column there is the respective rank in HPL, according to Top500. It is fundamental to observe that each machine in this brief list is equipped with accelerators, since the Japanese Fugaku cannot catch up with accelerators' energy efficiency per $Flop$. This is the main reason why heterogeneous computing, in which different hardware technologies are combined together to build each computing node, will be dominant in the future. In particular, accelerated platforms have multi-purpose CPUs with low computing throughput, whose purpose is to manage the accelerators' workload, still doing computations when very specific libraries have not been accelerated yet, i.e. Fourier Transforms or communication libraries, which are not yet available in all the platforms. We notice the presence in this classification of the Australian Setonix-GPU HPC machine, which has been used for the tests that will be discussed in Chapter 6. In particular, the machine has been thought to be as green as possible, being partially fed with solar panels and cooled with the aquifer water behind Perth. Furthermore, people there put all the available counters, for both CPUs and accelerators at our disposal, in order to help us with our energy measurements. Another important feature of the machine is the capability to change the CPU frequency at each run, in order to test the energy impact and the performance degradation due to the clock frequency reduction.

Following the discussion of Section 1.2, there are two machines who have reached the exascale in HPL, however, as we stressed out, HPL is not a good reference for true scientific codes, especially for the high-resolution simulations needed in numerical cosmology and condensed matter physics. A deep observation of Figure 1.4 and Figure 1.5 shows that, taking as reference the Fugaku supercomputer, and the American Frontier and Aurora supercomputers, a trivial calculation implies that reaching exascale in HPCG means consuming $837MW$ for Fugaku, $1.96GW$ for Frontier and the enormous number of $7GW$ for Aurora. These numbers are incredibly huge, especially for accelerated machines, which would need large nuclear power plants (one per ma-

[5] https://top500.org/lists/green500/2024/06/

chine) to be fed in order to reach the "real" exascale. It is easy to understand that such a huge amount of nuclear fuel would lead to outstanding cost of production and, much more important, to environmental disasters. To shed some light on this dark future, both companies and computer scientists are collaborating to develop always greener hardware technologies and, in the meantime, algorithms which suit to these new technologies, aware to find the cheapest instructions at the assembler level.

From the hardware's point of view much effort has been done to develop more green technologies, like the new ARM (Advanced Risc Machine) CPUs, whose complexity is much smaller compared to Intel and AMD CPUs, which reduce the energy-to-solution while maintaining a good time-to-solution (Aroca et al., 2012; Calore et al., 2020; Noor Rahman et al., 2024; Suárez et al., 2024; Dakić et al., 2024). However, in the Green500 list, the first totally ARM machine is ranked only $61^{th}$. This happened because HPC has been going through heterogeneous computing, which includes multi-core CPUs and GPUs (Graphic Processing Units).

GPUs, whose scheme is compared to the CPU one in Figure 1.7, have been thought for graphic purposes, which needed massive parallelism to construct images whose pixels were independent from one another. In early 2000's they started to be used in HPC as well, because of their enormous natural parallelism. They have up to thousands of computing cores, which are singularly smaller than multi-purpose CPU cores, but are incredibly efficient for computing, i.e. they have logic units and instructions for summations, multiplications and transcendental functions as well. Of course, unlike graphic problems, actual scientific codes need cores' synchronization, because usually the computation which is assigned to a specific GPU core does depend on the ones assigned to other cores, so kind of barriers are needed in the codes. However, in many cases, GPUs turn out to be much faster than CPUs. They cannot be installed directly on a machine without CPUs, however, because they need a central processing unit to give them the right instructions. Unless with the new unified architecture schemes, like NVIDIA Grace Hopper 200 chip[6] or AMD Radeon Instict MI300X APU[7], memory exchanges between CPUs and GPUs are needed because the GPU cannot directly access the CPU memory. The accelerators then store data in their RAM memory, but they do have some cache levels as well, each one shared among smaller and smaller numbers of GPU cores.

GPU energy efficiency has been studied in (Huang et al., 2009; B. Wang et al., 2013; Mittal et al., 2014; Qasaimeh et al., 2019; Jahanshahi et al., 2020), which confirm that heterogeneous CPU+GPU computing will be prevalent in the future architectures. GPU computing will be the milestone of this thesis, as will be clear in the next chapters.

In Figure 1.5, it is noticeable that GPUs perform better for HPL tests rather than HPCG ones. The main explanation for this gap of performance is due to the memory imprint of HPCG, which, as already mentioned, impacts significantly on the result. To sum up, in the case of solving linear equation systems, GPUs appear much faster

[6] `https://www.nvidia.com/it-it/data-center/grace-hopper-superchip/`
[7] `https://www.amd.com/en/products/accelerators/instinct/mi300/mi300x.html`

than CPUs, but when communication becomes relevant GPU machines start to suffer a deeper performance drop. However, to face this problem, current GPU vendors are developing libraries to handle memory dominated applications, available in HPC-SDK[8] for NVIDIA, ROCm[9] for AMD and oneMKL[10] for Intel. These communication schemes are extremely capable to fix this memory issue, making memory dominated algorithms to run efficiently on GPUs as well. Their impact compared to standard CPU cases will be presented and discussed in Chapter 4,Chapter 5,Chapter 6.

For the sake of completeness, we include FPGAs (Field Programmable Gate Array) in this Section. An FPGA is not an hardware with fixed architecture, but it's a "box" with a set of logic units inside which are activated depending on the code's purposes. This makes them really not user-friendly and sometimes it's hard to compile a code because the set of logic units in the device doesn't match with the code instructions. Their energy efficiency has been studied in (Betkaoui et al., 2010; Qasaimeh et al., 2019; Nguyen et al., 2020; Nguyen et al., 2022). In some extent, they're more efficient in trade-off between energy-to-solution and time-to-solution than GPUs. An FPGA scheme example is shown in Figure 1.8.

In terms of atomic operations, much effort has been put in order to improve the silicon chips to save as much energy as possible. Figure 1.9, from the update about (Leland et al., 2014) work, shows how the energy cost, measured in picojoules $pJ$, changed from 2012 to 2020. In particular, a double-precision floating point operation used to consume $25pJ$, which became $4pJ$ in 2020. All the other operations involve memory movements, i.e. energy spent to access CPU registers, to access memory regions far $1-5mm$, to go off the silicon chip, and eventually to access DRAM and global memory. The take-home message in this plot is that arithmetic intensive operations are much cheaper than memory operations. This means that the greenest solution is to write codes whose arithmetic intensity is much larger than memory intensity, especially for GPU applications. However, this is not always possible, and many memory dominated applications do exist, whose memory impact dramatically increases the larger computing resources used. In these cases it becomes fundamental to modify the memory intense algorithms to reduce as much as possible the memory operations, for instance saving them in registers to avoid memory access at each step when variables are repeatedly used.

[8] `https://developer.nvidia.com/hpc-sdk`
[9] `https://www.amd.com/en/products/software/rocm.html`
[10] `https://www.intel.com/content/www/us/en/developer/tools/oneapi/onemkl.html`

| Rank | TOP500 Rank | System | Cores | Rmax (PFlop/s) | Power (kW) | Energy Efficiency (GFlops/watts) |
|---|---|---|---|---|---|---|
| 1 | 189 | **JEDI** - BullSequana XH3000, Grace Hopper Superchip 72C 3GHz, NVIDIA GH200 Superchip, Quad-Rail NVIDIA InfiniBand NDR200, ParTec/EVIDEN EuroHPC/FZJ Germany | 19,584 | 4.50 | 67 | 72.733 |
| 2 | 128 | **Isambard-AI phase 1** - HPE Cray EX254n, NVIDIA Grace 72C 3.1GHz, NVIDIA GH200 Superchip, Slingshot-11, HPE University of Bristol United Kingdom | 34,272 | 7.42 | 117 | 68.835 |
| 3 | 55 | **Helios GPU** - HPE Cray EX254n, NVIDIA Grace 72C 3.1GHz, NVIDIA GH200 Superchip, Slingshot-11, HPE Cyfronet Poland | 89,760 | 19.14 | 317 | 66.948 |
| 4 | 328 | **Henri** - ThinkSystem SR670 V2, Intel Xeon Platinum 8362 32C 2.8GHz, NVIDIA H100 80GB PCIe, Infiniband HDR, Lenovo Flatiron Institute United States | 8,288 | 2.88 | 44 | 65.396 |
| 5 | 71 | **preAlps** - HPE Cray EX254n, NVIDIA Grace 72C 3.1GHz, NVIDIA GH200 Superchip, Slingshot-11, HPE Swiss National Supercomputing Centre (CSCS) Switzerland | 81,600 | 15.47 | 240 | 64.381 |
| 6 | 299 | **HoreKa-Teal** - ThinkSystem SD665-N V3, AMD EPYC 9354 32C 3.25GHz, Nvidia H100 94Gb SXM5, Infiniband NDR200, Lenovo Karlsruher Institut für Technologie (KIT) Germany | 13,616 | 3.12 | 50 | 62.964 |
| 7 | 54 | **Frontier TDS** - HPE Cray EX235a, AMD Optimized 3rd Generation EPYC 64C 2GHz, AMD Instinct MI250X, Slingshot-11, HPE DOE/SC/Oak Ridge National Laboratory United States | 120,832 | 19.20 | 309 | 62.684 |
| 8 | 11 | **Venado** - HPE Cray EX254n, NVIDIA Grace 72C 3.1GHz, NVIDIA GH200 Superchip, Slingshot-11, HPE DOE/NNSA/LANL United States | 481,440 | 98.51 | 1,662 | 59.287 |
| 9 | 20 | **Adastra** - HPE Cray EX235a, AMD Optimized 3rd Generation EPYC 64C 2GHz, AMD Instinct MI250X, Slingshot-11, HPE Grand Equipement National de Calcul Intensif - Centre Informatique National de l'Enseignement Suprieur (GENCI-CINES) France | 319,072 | 46.10 | 921 | 58.021 |
| 10 | 28 | **Setonix – GPU** - HPE Cray EX235a, AMD Optimized 3rd Generation EPYC 64C 2GHz, AMD Instinct MI250X, Slingshot-11, HPE Pawsey Supercomputing Centre, Kensington, Western Australia Australia | 181,248 | 27.16 | 477 | 56.983 |

**Figure 1.6:** *First* 10 *machines in the Green500 list. Source:* `https://top500.org/lists/hpcg/2024/06/`.

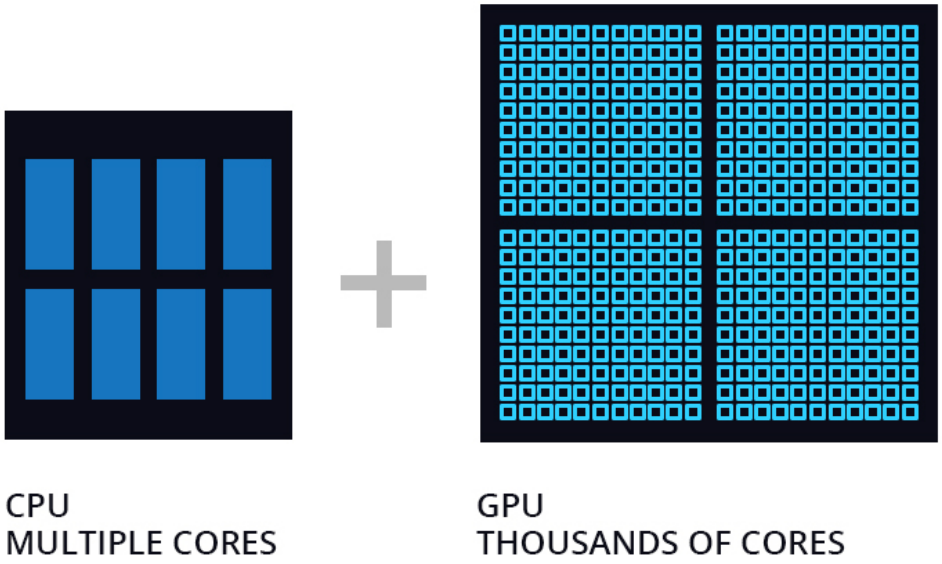



**Figure 1.7:** *CPU and GPU schemes, respectively. CPUs have large multi-purpose cores, whereas GPUs have smaller cores specific for massively parallel computing. Source:* `https://community.fs.com/it/article/deep-comparison-between-server-cpu-and-gpu.html`

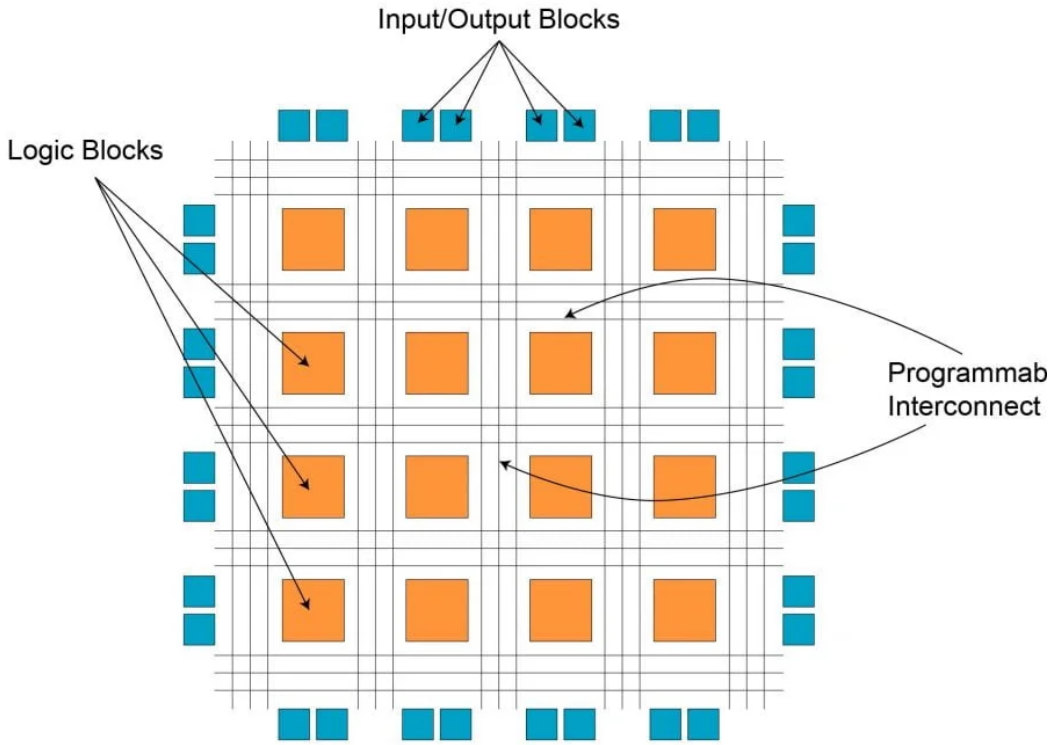



**Figure 1.8:** *Schematic example of an internal structure in an FPGA. Source:* `https://www.logic-fruit.com/blog/fpga/fpga-design-architecture-and-applications/`

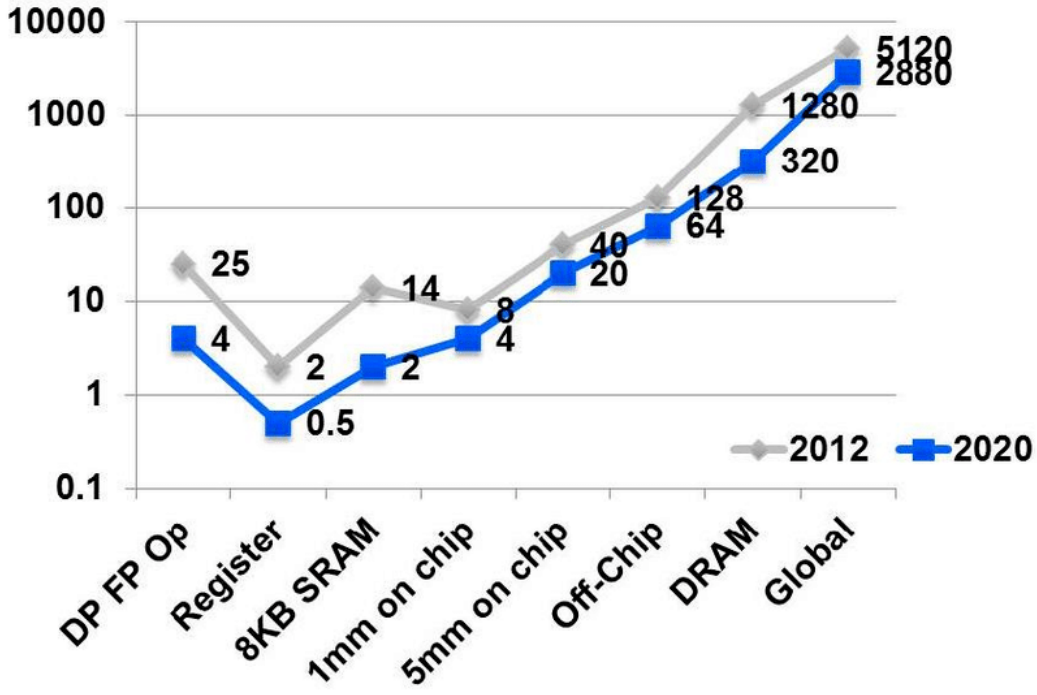



**Figure 1.9:** *Energy cost, in picojoules (pJ) per 64-bit floating-point operation, for various common operations within a computer. The upper (gray) line characterizes energy cost estimates in 2012 technology, and the lower (blue) line projects costs in 2020. Note that the double-precision floating-point arithmetic (DP FP Op) energy cost is comparable to that for moving the same data 1mm–5mm on chip; that cost is dwarfed by the cost of any movement of this same data off chip. Source: (Leland et al., 2014) update.*

## 1.4 Parallel computing

Moore's law is an empirical law stating that the number of transistors in an integrated-circuit doubles every two years, at the beginning, and every 18 months with technological process, while the power consumption remains constant (G. Moore, 1998; G. E. Moore, 2006). This last result is due to Dennard law (Dennard et al., 1974; Dennard et al., 1999):

$$P \propto C \times V^2 \times f \tag{1.1}$$

$C$ is the transistor capacitance, scaling as the area, $V$ is the voltage, scaling as the linear dimension and $f$ is the frequency. If the linear size shrinks so do the voltage, meaning that if the area remains equal, the frequency can increase without impacting the power consumption.

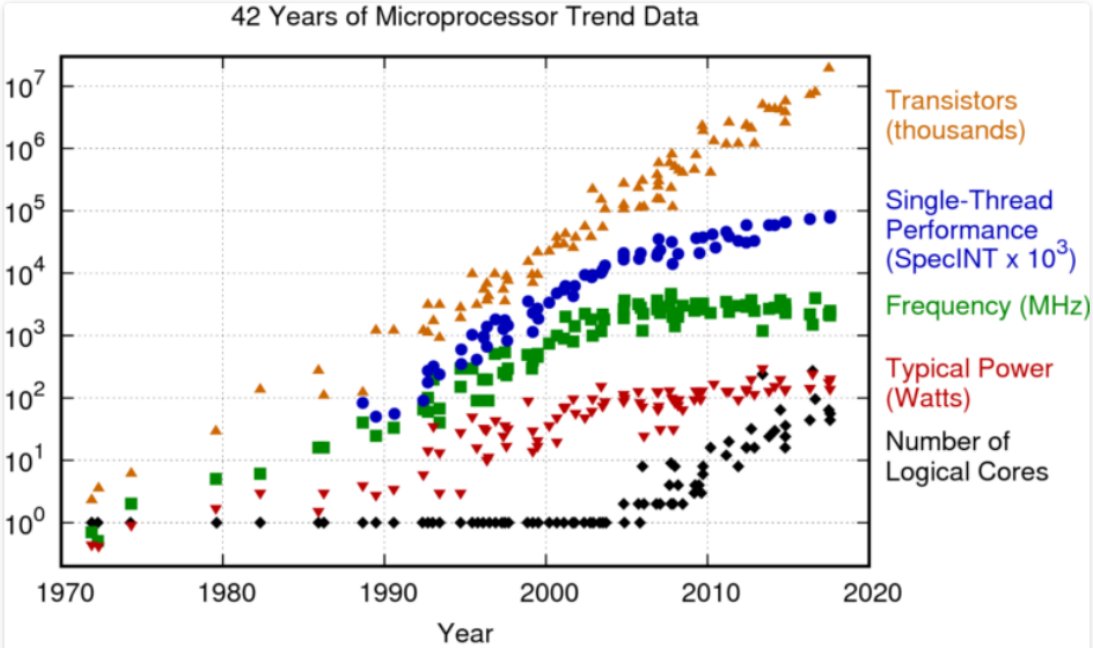



**Figure 1.10:** *Original data up to the year* 2010 *collected and plotted by M.Horovitz, F.Labonte, O.Shacham, K.Olukotun, L.Hammond, and C.Batten.*
*New plot and data collected for* 2010 – 2017 *by K.Rupp.*

However, Figure 1.10, where the number of *GFlops* is plotted as a function of the year, shows that even if the number of transistor still follows the Moore's law, single-thread performance and CPU frequency have a worse increase than expected, whereas power consumption started to increase in 1990's. This happens because when the scales become too small, quantum effects start to be dominant. We can summarize them into:

- Leakage current
- Threshold voltage
- Physical limits at atomic scales

So, as transistors get smaller, power consumption actually increases, leading to a **power wall** which prevented the CPU frequency to increase beyond $3GHz$ since the last decade. The question is: why does the transistor number still follows the Moore's law?

Figure 1.10 shows that another quantity, i.e. the number of logical cores, increases as well. Until early 2000's CPUs were *serial*, i.e. they were constituted by a single core only. People thought that if quantum effects destroyed Dennard scaling (Esmaeilzadeh et al., 2011), the increase in CPU computing power could still be achieved by increasing the number of CPU cores. This trend lead from dual-core CPUs up to current $\sim 10^2$ cores CPUs.

Two CPU cores of $3GHz$ each do **not** have the same computing power of one $6GHz$ core, due to:

- Cores coordination (at hardware level)
- Threads coordination (at software level)
- Memory contention access
- Increased algorithmic complexity

Figure 1.11 shows a simplified example of an Intel CPU with 8-cores (Zajqc et al., 2018). Each core has a level 1 cache memory, L1, which has a few $KB$ capacity and its owned by the specific core only. It is the fastest memory to access. Each core has also L2 cache, with hundreds of $KB$ capacity in general, which can be either core's exclusive or shared among more than one core. Shown in the figure there is L3 cache, shared among all the cores and with up to $MB$ of memory capacity. L3 is the slowest cache level to access (Saavedra et al., 1995; Henning, 2000; Pas, 2002). Eventually there is the RAM (Random Access Memory), shared over all the cores, which has in general tens of $GB$ capacity and is by far the slowest memory accessible by the cores.

Multi-core CPUs pioneered the era of parallel computing. Before that, parallel computing was made by connecting two or more computers with single-core CPUs. Although several attempts have been made to reduce the programmers' workload and to make parallel computing user-friendly, it has been very hard to write compilers which take as input serial codes and parallelize them. The only efficient way to handle this problem is to use parallelization paradigms, which permit the programmer to rely on libraries, available in C,C++,Fortran,Python and other languages, to implement parallel codes. The parallelization paradigms are split into:

- **distributed-memory**, where parallel workers are given only memory portions, and cannot directly access to memory portions owned by other workers.
- **shared-memory**, where parallel workers share the entire CPU memory.

These two concepts are fundamental in parallel programming, and will be discussed several times in this thesis. The two most popular ones for each paradigm are:

- **MPI**, Message Passing Interface.

- **OpenMP**, Open MultiProcessing[11].

Almost each C/C++ compiler supports OpenMP (Chandra et al., 2001; Chapman et al., 2007; Pas et al., 2017), which is a *pragma-based* paradigm, it allows the programmer to include parallelism by impacting as little as possible on the code.

In Figure 1.12 there is an example of a single-thread "*Hello world*". Because it's the serial version, no *pragma* is needed, and the function call identifying the thread recognizes only process 0. The situation becomes more interesting in Figure 1.13, when a parallel region is opened with the *pragma*. In this case several threads are spawned, depending on the environment variable declared in the central panel. By setting the number of OpenMP threads to 4, 4 different processes will print "*Hello world*".

MPI is a standard for distributed parallel computing, where each MPI process (or rank of task), owns a specific memory region and cannot access memory pertaining to another process. For a more detailed discussion about MPI and its functions see Chapter 3. The first version, MPI-1, was released in June 1994, after almost three years of efforts. In the next decades, MPI-2, MPI-3 and MPI-4 were released, including at each step new features including the possibility of using MPI processes in shared-memory and non-blocking MPI functions (see Chapter 3) (Walker, 1992; The MPI Forum, 1993; Sur et al., 2006; Nielsen, 2016). MPI-5 version is currently under development. There are several implementations of these standards, like OpenMPI[12], MPICH[13], MVAPICH[14], IntelMPI[15], SpectrumMPI[16].

In Figure 1.14 we show an example of "*Hello world*" code in MPI. At first glance, we note that MPI introduces a new level of complexity compared to OpenMP, which is *pragma-based*, with the inclusion of MPI functions. The code is run with *mpirun/mpiexec/srun* calls, and the number of MPI processes is chosen with *-n/-np N* by the programmer during each execution. No environmental variables and code modifications are needed in this case to run the code with $1, 2, 3, 4, \ldots$ MPI tasks, respectively.

It is important to say that the code shown in Figure 1.14 is exactly the same and works perfectly in any MPI implementation, meaning that in this case an MPI code is portable. However, when more complex and exotic functions are used in MPI, for instance in the new communication implementation described in Chapter 3, one out of the preceding MPI implementations can display errors at compile time or runtime, and if not, important performance differences are usually present. This happens because although the functions have the same names in any MPI library, the underlying implementations may have important differences, i.e. there are several ways to write a C code in MPI for a specific function.

[11] https://www.openmp.org/
[12] https://www.open-mpi.org/
[13] https://www.mpich.org/
[14] https://mvapich.cse.ohio-state.edu/
[15] https://www.intel.com/content/www/us/en/developer/tools/oneapi/mpi-library-documentation.html
[16] https://www.ibm.com/it-it/products/spectrum-mpi

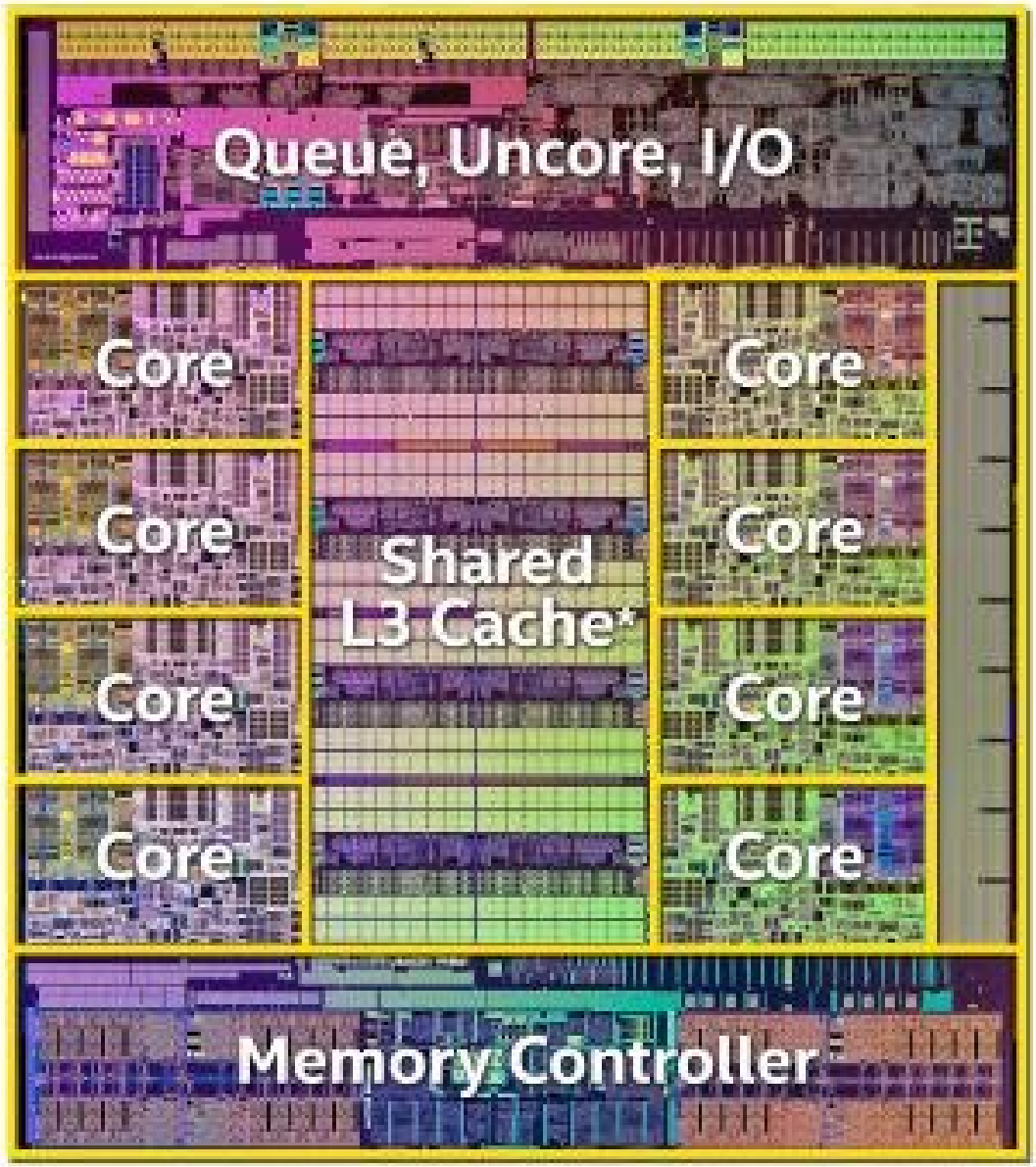



**Figure 1.11:** *Simplified example of an Intel CPU with 8-cores, the L3 cache shared among all the cores. Source:* (*Zajqc et al., 2018*).

```c
#include <stdio.h>
#include <omp.h>

int main(int argc, char** argv){

    printf("Hello from process: %d\n", omp_get_thread_num());

    return 0;
}
```

```
Hello from process: 0
```

**Figure 1.12:** *"Hello world" OpenMP C code and corresponding output without opening any parallel region. Source:* `https://curc.readthedocs.io/en/latest/programming/OpenMP-C.html`

```c
#include <stdio.h>
#include <omp.h>

int main(int argc, char** argv){
    #pragma omp parallel
    {
        printf("Hello from process: %d\n", omp_get_thread_num());
    }
    return 0;
}
```

```
export OMP_NUM_THREADS=4
```

```
Hello from process: 3
Hello from process: 0
Hello from process: 2
Hello from process: 1
```

**Figure 1.13:** *"Hello world" OpenMP C code and corresponding output with a parallel region and exporting the number of OpenMP threads. Source:* `https://curc.readthedocs.io/en/latest/programming/OpenMP-C.html`

```c
#include <mpi.h>
#include <stdio.h>

int main(int argc, char** argv) {
    // Initialize the MPI environment
    MPI_Init(NULL, NULL);

    // Get the number of processes
    int world_size;
    MPI_Comm_size(MPI_COMM_WORLD, &world_size);

    // Get the rank of the process
    int world_rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);

    // Get the name of the processor
    char processor_name[MPI_MAX_PROCESSOR_NAME];
    int name_len;
    MPI_Get_processor_name(processor_name, &name_len);

    // Print off a hello world message
    printf("Hello world from processor %s, rank %d out of %d processor
s\n",
           processor_name, world_rank, world_size);

    // Finalize the MPI environment.
    MPI_Finalize();
}
```

```
/home/kendall/bin/mpirun -n 4 -f host_file ./mpi_hello_world
Hello world from processor cetus2, rank 1 out of 4 processors
Hello world from processor cetus1, rank 0 out of 4 processors
Hello world from processor cetus4, rank 3 out of 4 processors
Hello world from processor cetus3, rank 2 out of 4 processors
```

**Figure 1.14:** *"Hello world" MPI example C code and corresponding output with 4 MPI processes. Source: https://mpitutorial.com/tutorials/mpi-hello-world/.*

## 1.5 GPU computing

As already mentioned in Section 1.3, the employment of Graphic Processing Units for scientific computing started in early 2000's (Krüger et al., 2003; Bolz et al., 2003; Tarditi et al., 2006; Owens et al., 2007; Owens et al., 2008).

However, at the beginning GPU computing was very hard, because programmers had to rely on really low-level languages, like OpenGL[17] (Segal et al., 2004) and DirectX[18], which were not user-friendly because of their cumbersome API (Application Programming Interface). General Purpose GPU computing (GPGPU) became more popular when NVIDIA introduced its new own language, CUDA (Compute Unified Device Architecture)[19] (Sanders et al., 2010) (see (Manavski et al., 2008) as one of the first examples of CUDA scientific computing). CUDA is a C++ extension which contains new APIs to allow GPU programming. A standard CUDA program is split into:

- **APIs**, which are functions called by the CPU whose purpose is to manage the CPU (host) and the GPU (device) interations. Several functions are available to allocate memory on the GPU, to copy memory from host to device and vice-versa, to synchronize both the host and the device, and so on.
- **Kernels**, which are the functions that are actually "offloaded".

Kernels can be divided their turn into:

- **host**: this function can only be called by the CPU and is executed by the CPU itself.
- **global**: this function can only be called by the CPU and is executed by the GPU.
- **device**: this function can only be called inside a global kernel by the GPU and is executed by the GPU itself.

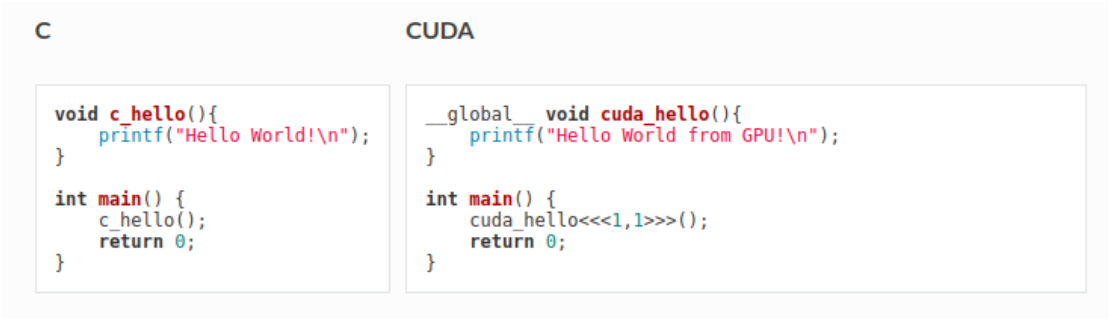



**Figure 1.15:** *Differences in an "Hello world" code in C and CUDA. Source:* `https://krutikabapat.github.io/Learn-CUDA-Programming-using-Google-Colab/`.

Figure 1.15 is an example in which the differences between C and CUDA are shown for an "Hello world" code. We see the two functions that are called, which is a standard

[17] `https://www.opengl.org/`
[18] `https://www.microsoft.com/it-it/download/details.aspx?id=3`
[19] `https://developer.nvidia.com/cuda-toolkit`

*void* for the C code, while it has that *global* clause before *void*. The two numbers between the angle brackets in the right panels are the number of block of threads that we want to use and the number of threads per each block. This is because of the underlying hardware, which is made of several **streaming multiprocessors**, that can be thought roughly as groups of cores. For the sake of completeness, all the *global* functions must be void and cannot return anything to the host. In this case GPU natural parallelism is not exploited, since just one block with one threads is actually running.

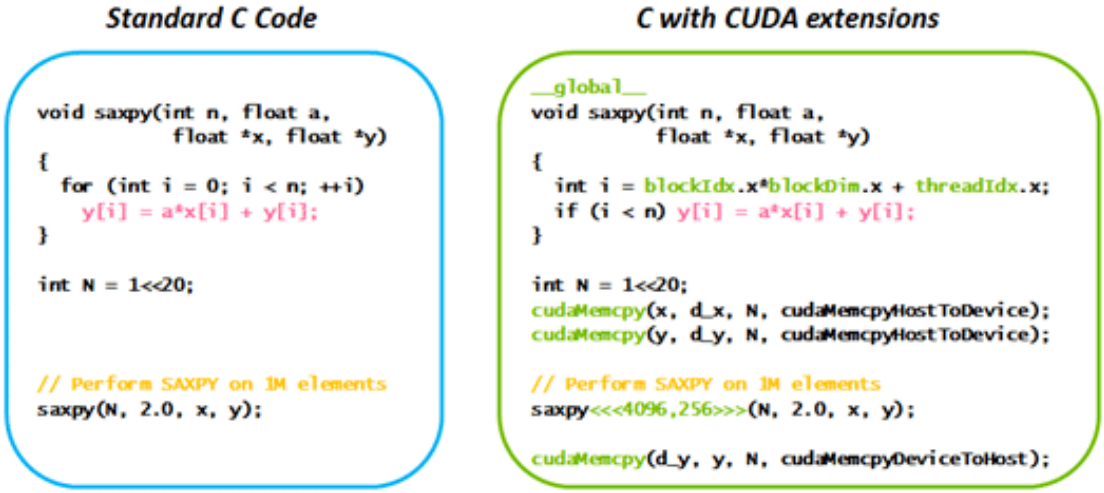



**Figure 1.16:** *Saxpy operation, in which one vector is multiplied by a constant and it's summed to a second vector to obtain a final vector. The C example (left panel) and the CUDA example (right panel) are shown. Source:* `https://blogs.nvidia.com/blog/what-is-cuda-2/`.

In Figure 1.16 the comparison between a C saxpy code and the corresponding CUDA one is shown. Saxpy is a test in which there are two input vectors, where the first one is multiplied by a constant and then summed to the second one to get the resulting vector. Differently from the previous case, now it is necessary to transfer memory between the CPU and the GPU, because at the beginning the original vectors are only existing on the host. The function is called *cudaMemcpy*. The first argument is the new buffer, while the second argument is the original vector that the programmer wants to copy. The third argument is the array length, while the last one is the direction of the operation, i.e. we can copy both from CPU to GPU or from GPU to CPU. The first two calls are the copies of the two vectors onto the device. After that, the saxpy function is called, this time with many more GPU threads, in the particular case $N_{blocks} = 4096$ and $N_{threads} = 256$, for a total of $N_{tot} = 1048576$ total threads. If the array dimension, here labelled with $N$ or $n$ is greater than $N_{tot}$, the *if-clause* in the *global* function says that only the first $n$ threads are actually performing the operation. Eventually, the updated vector is copied back to the host which gets the final result.

However, CUDA is not the only language extension to program a GPU. For instance, AMD GPUs are programemd with another language, whose calls are exactly equal to CUDA ones, which is called HIP (Heterogeneous-computing Interface for

Portability)[20]. HIP (Sun et al., 2022) is included in the AMD software stack, ROCm[21]. The only observable difference of a HIP code with the one shown in Figure 1.16 is the replacement of the *cudaMemcpy* function call with *hipMemcpy*. Another relevant difference regards the number of threads per block, that we called $N_{threads}$ above, which must be a multiple of 32 in the case of NVIDIA GPUs, while must be a multiple of 64 in the case of AMD. This is due to hardware differences. This minimum thread number is called **warp size**.

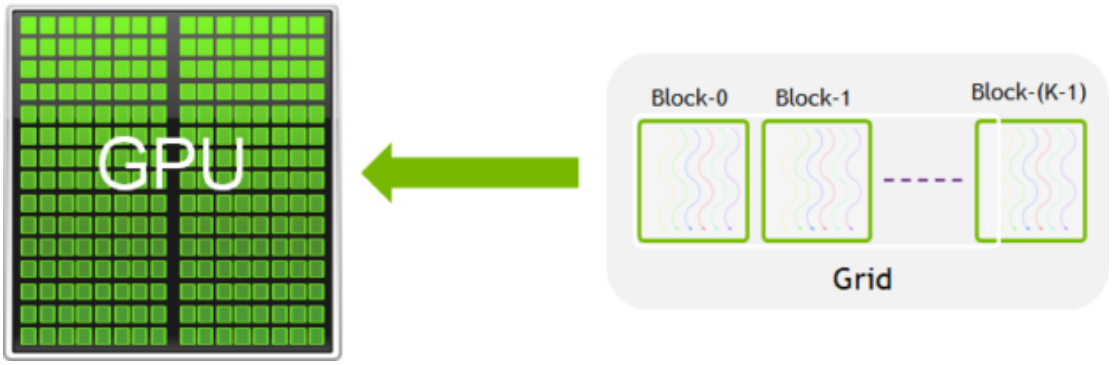



**Figure 1.17:** *CUDA kernels are subdivided into blocks. Source:* `https://developer.nvidia.com/blog/cuda-refresher-cuda-programming-model/`.

Figure 1.17 shows an example of how GPU cores can be assigned to different thread blocks. As discussed above, it is necessary to find the best trade-off between the hardware requirements, i.e. the minimum warp size, and the code requirements, i.e. the thread displacement depending on the code. For example, when you have to perform vector operations, standard linear thread assignment is the best solution, while it is not the case for matrix operations, in which the topology may suggest to assign 2D blocks of threads, like the right panel in the figure.

CUDA is a low-level C++ extension for GPU programming, that requires a deep study of its syntax and very careful threads/blocks assignments, CUDA calls and kernel writing to make the code reach the best performance available. However, there are other *pragma-based* paradigms, i.e. OpenACC[22] (Wienke et al., 2012; Herdman et al., 2014; Sabne et al., 2015; Farber, 2016; Chandrasekaran et al., 2017; Oyarzun et al., 2021), which has been used even for FPGA computing (S. Lee et al., 2016), and OpenMP[23] (S. Lee et al., 2009; S. Lee et al., 2010; Bertolli et al., 2014; Hayashi et al., 2019; Huber et al., 2022; Doerfert et al., 2022), which allow for GPU offloading.

In Figure 1.18 we show the GPU porting on OpenACC of the 2D Laplace solution in Fortran. The original equation is:

$$\Delta f(x,y) = \frac{\partial^2 f(x,y)}{\partial x^2} + \frac{\partial^2 f(x,y)}{\partial y^2} = 0 \tag{1.2}$$

[20] `https://rocm.docs.amd.com/projects/HIP/en/latest/`
[21] `https://www.amd.com/en/products/software/rocm.html`
[22] `https://www.openacc.org/`
[23] `https://enccs.github.io/openmp-gpu/`

The spatial discretization in the second-order scheme can be written as:

$$\frac{\partial^2 f(x,y)}{\partial x^2} = \frac{f(x_{i+1},y) - 2f(x_i,y) + f(x_{i-1},y)}{\Delta x} \tag{1.3}$$

Inserting Equation 1.3 into Equation 1.2 leads to the final expression:

$$f(x_i,y_j) = \frac{f(x_{i+1},y) + f(x_{i-1},y) + f(x,y_{j+1}) + f(x,y_{j-1})}{4} \tag{1.4}$$

Equation 1.4 is the equation actually solved in Figure 1.18. We notice the *data* constructs, where *copyin* means moving memory from the host to the device, while *copyout* means the opposite. Inside the parallel loop, there are the *gang,worker,vector* clauses, which are related to distribution of work among blocks (*gangs*), each one spawning threads (*worker*) with vectorization (*vector*). Vectorization is another level of parallelism in which the hardware's vector registers are used. They allow to perform more than one operation per clock, whose number depends on the length of the vector registers. The *collapse* clause is inserted to distribute among all the threads the nested loop cycle defined after. Reduction operation in the second parallel region means to find the maximum value of the error among all the threads.

Figure 1.19 is the corresponding OpenMP version of the code in Figure 1.18. The code structure is the same as the OpenACC case, there is just a syntax difference. In terms of data management, *copyin*/*copyout* are replaced with *mapto*/*mapfrom*, while *gang*/*worker*/*vector* are replaced with *teams*/*distribute*/*simd*, respectively. The *target* in the OpenMP version is what established that the next code lines need to run on the accelerator. Without the *target* clause a parallel region on the CPU as in the standard OpenMP case will be opened.

Although the two implementations are equivalent, OpenMP ensures much more portability than OpenACC, by having C/C++/Fortran support with several compilers for NVIDIA/AMD/INTEL GPUs, whereas OpenACC has C/C++/Fortran support just for NVIDIA GPUs, and a Fortran support for AMD GPUs. If we want our code to run onto different HPC platforms, the best solution is to use OpenMP. In conclusion, both OpenACC and OpenMP are much more user-friendly than CUDA, because they allow you to offload code portions on the GPU simply by adding few code lines, avoiding the need to write CUDA kernels by hand and manage directly memory transfers.

```
                    **OpenACC**
!$acc data copyin(f) copyout(f_k)
   do while (max_err.gt.error.and.iter.le.max_iter)
!$acc parallel loop gang worker vector collapse(2)

      do j=2,ny-1
        do i=2,nx-1
           d2fx = f(i+1,j) + f(i-1,j)
           d2fy = f(i,j+1) + f(i,j-1)
           f_k(i,j) = 0.25*(d2fx + d2fy)
        enddo
      enddo
!$acc end parallel

       max_err=0.

!$acc parallel loop collapse(2) reduction(max:max_err)

      do j=2,ny-1
        do i=2,nx-1
           max_err = max(dabs(f_k(i,j)-f(i,j)),max_err)
           f(i,j) = f_k(i,j)
        enddo
       enddo
!$acc end parallel

       iter = iter + 1
    enddo
!$acc end data
```

**Figure 1.18:** *OpenACC code for 2D Laplace equation solution. Source: https://documentation.sigma2.no/code_development/guides/converting_acc2omp/openacc2openmp.htmlporting-openacc-to-openmp-offloading.*

```
                **OpenMP**
!$omp target data map(to:f) map(from:f_k)
   do while (max_err.gt.error.and.iter.le.max_iter)
!$omp target teams distribute parallel do simd collapse(2)
      schedule(static,1)
      do j=2,ny-1
        do i=2,nx-1
           d2fx = f(i+1,j) + f(i-1,j)
           d2fy = f(i,j+1) + f(i,j-1)
           f_k(i,j) = 0.25*(d2fx + d2fy)
         enddo
       enddo
!$omp end target teams distribute parallel do simd

        max_err=0.

!$omp target teams distribute parallel do simd collapse(2)
       schedule(static,1) reduction(max:max_err)
      do j=2,ny-1
        do i=2,nx-1
           max_err = max(dabs(f_k(i,j)-f(i,j)),max_err)
           f(i,j) = f_k(i,j)
        enddo
      enddo
!$omp end target teams distribute parallel do simd

      iter = iter + 1
   enddo
!$omp end target data
```

**Figure 1.19:** *OpenMP code for 2D Laplace equation solution. Source:* `https://documentation.sigma2.no/code_development/guides/converting_acc2omp/openacc2openmp.html#porting-openacc-to-openmp-offloading`.

## 1.6 Strong and weak scalability

The performance in parallel codes is measured in terms of the scalability, which describes the change in the code's runtime when computing resources or problem sizes increase. In particular, the relevant quantities associated to codes' scalability are the **speed-up** or the **efficiency**, defined as:

$$Sp = \frac{T_0}{T_N}; \qquad Eff = \frac{T_0}{N \times T_N} \tag{1.5}$$

In Equation 1.5, $T_0$ is the lowest computing resources runtime, which is for instance the serial case, while $T_N$ is the code's runtime with $N$ parallel workers. In the particular case of **strong scalability**, i.e. the problem size is fixed but the number of parallel workers increases, the ideal theoretical solution is:

$$Sp = N; \qquad Eff = 1 \tag{1.6}$$

This ideal case hardly ever happens because almost each parallel code is composed by a serial and a parallel part. Thus, to compute the actual speedup achievable, the correct version of Equation 1.5 is given by the Amdahl's law (Amdahl, 1967):

$$Sp = \frac{1}{(1-P) + \frac{P}{N}} \tag{1.7}$$

In Equation 1.7, $P$ is the parallel fraction of the code. Of course, when $P = 1$, Equation 1.7 is equal to Equation 1.5.
Hereafter we considered only the case in which the problem size is fixed. However, this is not the general case, where we imagine that the problem size increase with the number of parallel workers. This is the case of **weak scalability**, and Amdahl's law needs to be generalized and becomes the Gustafson's law (Gustafson, 1988):

$$Sp = \frac{1}{D_N} \times \frac{1}{(1-P) + \frac{P}{N}} \tag{1.8}$$

$D_N$ in Equation 1.8 represents the problem size with $N$ workers. Thus, when $D_N = N$ we are doing weak scalability tests, whereas when $D_N = 1$, Equation 1.8 turns to Equation 1.7 and we are doing strong scalability tests.

## 1.7 The communication impact

The discussion of Section 1.6 holds for **embarassingly parallel** codes, where in the codes' parallel fraction each parallel worker is in charge to perform its own computation independently on what are doing all the other workers. However, this is not the actual case in scientific computing, in which each parallel worker needs information exchanges with the other workers. This is particularly true in radio astronomy imaging, where initial input data are distributed among workers in a time-ordered domain, and turning to a space-ordered domain requires each worker to exchange data with all the others, or in numerical cosmology codes, where workers need to exchange information about particles in boundaries of each domain.
Communication reduces the theoretical performance expected in Equation 1.7 and Equation 1.8 and its impact depends on the fraction between the communication time and the total time:

$$f = \frac{T_{comm}}{T_{tot}} \tag{1.9}$$

When $T_{comm} << T_{tot}$ the code is **computation-bound**, while starting from $T_{comm} \sim T_{tot}$ the code is **communication-bound**, leading to a significant speed-up degradation. When many computing resources are used, i.e. $N \to \infty$, $T_{comm} \to T_{tot}$ and $f \to 1$, meaning that the code is entirely dominated by the communication and increasing the number of workers does not lead to any speed-up.
Ideally, in this case the runtime remains constant by increasing $N$, but in reality there are several communication overheads, like communication latency, non-ideal network topology, allocation of buffers and so on, which lead to performance degradation instead of speed-up. When a code turns out to be strongly communication bound, the best solution is the one with the lowest $N$ fitting the memory requirements.
Mantaining a good communication-to-computation ratio is pivotal for the code scalability. When computing resources get larger but runtime remains constant (or becomes even worse), the time-to-solution is the same but the energy-to-solution increases, diminishing the code's efficiency.

## 1.8 What is radio imaging?

Radio interferometers measure information about the sky in Fourier space, also called visibility space. The relation between the visibility space measurement and the brightness distribution of astrophysical radio sources is described by the van Cittert–Zernike theorem (Born et al., 2013), which states that the two-point correlation function of the electric field measured by two antennas of a radio interferometer is the Fourier-transformed intensity distribution of the sources (Brunet et al., 2024):

$$V(u,v) = \int\int \frac{I(l,m)}{\sqrt{1-l^2-m^2}} \times e^{-2\pi i(ul+vm)} dldm \tag{1.10}$$

Where $I(l,m)$ is the image of the radio sky, $V(u,v)$ is the sampled visibility space, $l,m$ are the coordinates in the sky plane and $u,v$ are the coordinates in the interferometric plane. So far, the sky is considered as a plane, but actually it's a sphere. In Chapter 2 the full 3D problem will be explored.

By applying the inverse Fourier transformation to the data, one can reconstruct the image of the radio sky. First, the visibility measurements (which are complex data) are resampled onto a regular grid, using a gridding algorithm (Cornwell et al., 1981; Cornwell et al., 1992), for instance IDG (Van der Tol et al., 2018a).

Then, a 2D Fourier transform, i.e. a Discrete Fourier Transform (DFT), can be used to reconstruct the matrix $I(l,m)$:

$$I_{lm} = \sum_{j=0}^{Num-1} \sum_{k=0}^{Num-1} b_{jk} V_{jk} e^{-\frac{2\pi i}{Num}(jl+km)} \tag{1.11}$$

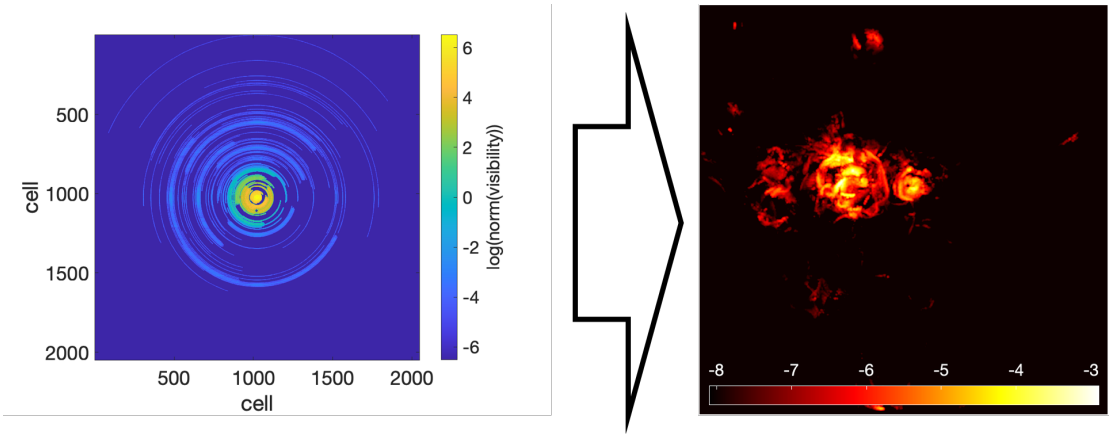



**Figure 1.20:** *Raw data in the visibility space need to be Fourier transformed to reconstruct a sky image in the real space.*

In Equation 1.11 $b_{jk}$ is the visibility sampling factor defined by the instrument geometry. Actually, $I$ is in this case a "dirty image", and the true image is obtained as $I_{CLEAN} = B \times I$, where $B$ is the point spread function, or the "dirty beam" of the instrument, and is defined as the Fourier transform of $b$. The true sky image $I_{CLEAN}$ must

then be constructed by recursively deconvolving the dirty beam from the dirty image $I$ with a process like such as the CLEAN algorithm (Högbom, 1974).

An example is shown in Figure 1.20, where raw visibility data, in the left panel, displaying "trails", need to be transformed from Fourier to real space in order to reconstruct the sky image. In the figure, the right panel image has already been deconvolved.

**Figure 1.21:** *Optical analogue of trails in visibility space. This is due to Earth rotation during each observation. Source: Shutterstock.*

In order to make "trails" more easy to understand, Figure 1.21 is an optical analogue of the left panel of Figure 1.20. During observations, which last several hours, the Earth rotation forces the survey to follow the trail of each sky source.

By applying the Discrete Fourier Transform to pointlike visibilities, i.e. with Equation 1.11, we get the sky brightness. However, the algorithm is extremely slow, by having $Num^2$ dependency, so the best choice is to utilize Fast Fourier Transform (FFT) to the original data in order to transform them to the real space (Heideman et al., 1984). This algorithm has a "smoother" $Num \log Num$ dependency, by exploiting symmetries in mathematical analysis. FFT can be applied only on regular meshes, two-dimensional for 2D FFT and three-dimensional for 3D FFT. This forces people to implement the gridding algorithm (Sault et al., 2007; Ye et al., 2020), which convolves pointlike visibilities with a kernel function and assigns them to grid points. Once visibilities are gridded, FFT can be applied and the final result is obtained.

This two steps of the radio imaging pipeline require a huge CPU time when running in a single CPU core, i.e. in serial. The solution to speedup the pipeline is to rely on High Performance Computing (HPC), by exploiting the code parallelization. Parallelization means that many parallel workers, which in this case are all the CPU cores available, each one being assigned a smaller computational domain while, in the

meantime, all the other workers perform computations in all the other domains. In the next sections of this chapter and in Chapter 3, the history of parallel computing will be explored and instructions on how to parallelize simple codes will be given, in order to have all the instruments to understand then how our new radio imaging code works.

## 1.9 HPC and radio astronomy

The new radio-interferometers, like the LOw Frequency ARray (LOFAR, van Haarlem et al., 2013), MeerKAT (Jonas et al., 2016), the Murchison Widefield Array (MWA, Mitchell et al., 2010), the Australian Square Kilometre Array Pathfinder (ASKAP, Johnston et al., 2007), will be precursors of what will be delivered by the Square Kilometre Array (SKA[24]), which will have two stations, one in Australia, SKA-Low, with observations in the frequency range $50-350MHz$, and another one in South Africa, SKA-Mid, with observations in the frequency range $350MHz - 15.4GHz$.

In particular, SKA-Low, whose precise location is the Murchinson desert, in Western Australia (WA), is composed of 131072 antennas with Christmas tree shape, covering a total area of $0.42km^2$. Figure 1.22 is a picture of the SKA-Low antennas with their singular Christmas-tree shape.

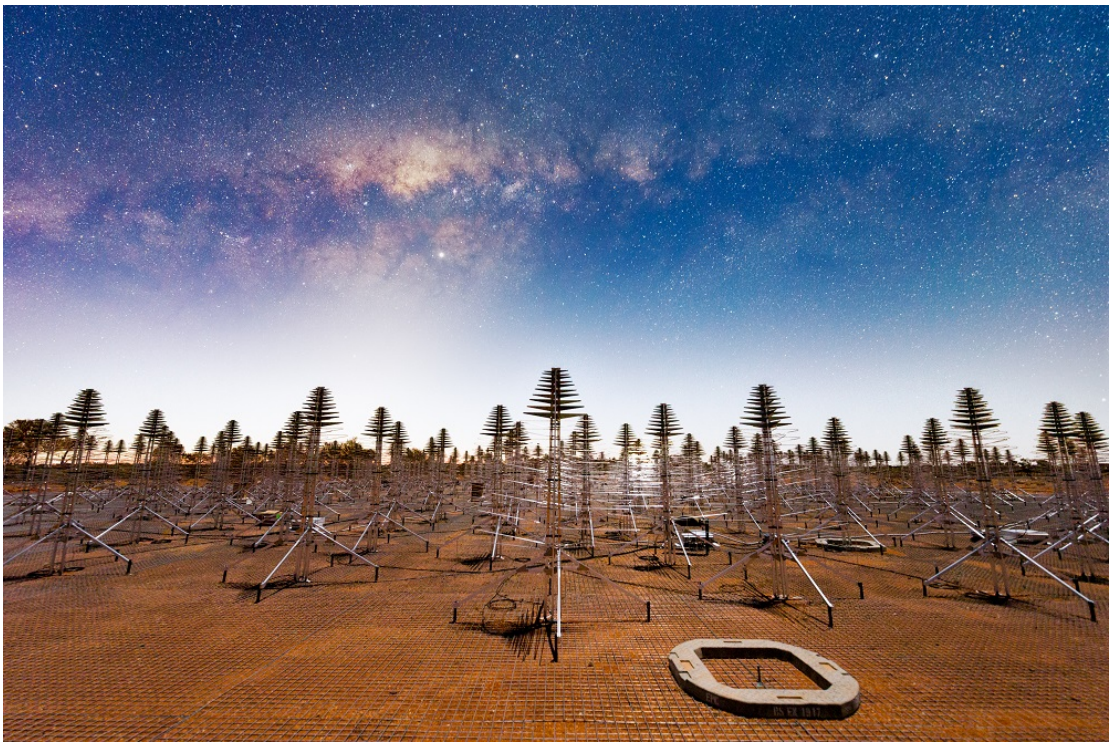


**Figure 1.22:** *20-second exposure showing Milky Way above the SKA-Low antennas. Credit: Michael Goh/ICRAR/Curtin. Source:* `https://spaceaustralia.com/news/ska-low-prototype-used-radio-transient-detections`.

[24] `https://www.skatelescope.org/`

The antennas are distributed in 512 stations, and when all the stations will start to survey the sky, around $\sim 1TB/s$ of data will be delivered, turning to $\sim 300PB/year$. These exceptional amount of data need to be analysed and processed. Data must be deleted from storage, because currently available machines don't have such a huge number of storage capacity, and there's the possibility that much data will be lost if not processed in time.

For these reasons, what is currently called **edge-computing** will be necessary:

- It will be useful to have the actual HPC machine the closest possible to the interferometers, to avoid as much as possible large data transfers.
- Almost real-time computing will be unavoidable, because data will be continuously delivered and machines need to process them as fast as possible.

Raw data need to be calibrated and then turned to a measurement set, where each **visibility** will be assigned a coordinate in the $u, v, w$ space, i.e. the Fourier space in the reference frame of radio interferometers, a weight and real/imaginary parts, since they're complex visibilities. A more detailed description of visibility data will be found in Chapter 2.

However, this is only the initial part of the radio astronomy pipeline. What we will be focused on in this thesis will be the imaging pipeline, in which datasets are already calibrated and converted to measurement sets in order to be processed to obtain reconstructed images of the sky. There are already codes performing this pipeline, like CASA[25] (Jaeger, 2008; Emonts et al., 2019; Team et al., 2022) and WSClean[26] (A. R. Offringa et al., 2014; A. R. Offringa et al., 2017; Van der Tol et al., 2018b).

Both codes reproduce perfectly well the sky images, but they're performance on HPC machines is currently poor because they have not been fully parallelized. There are code portions which have been parallelized with MPI/OpenMP or ported to GPUs, but there are still relevant sequential parts which do impact in the overall code performance, as will be discussed in Section 1.6. Usually most of the runtime is spent in I/O (input/output) operations, i.e. when the measurement set is read from the file-system and in the writing of the final result, which starts to impact significantly when the image resolution gets high. This is not MPI parallelized, meaning that the total memory available in this situation is the node's memory. Memory limitations turn out to be relevant when very large input data need to be processed, forcing people to split data into smaller datasets, i.e. perform frequency/time splittings, in order to analyse and process smaller images at each step. This soon leads to I/O dominated algorithms. One possibility to avoid this bottleneck is the utilization of new fast algorithms for I/O operations, like ADIOS2[27] (William F. Godoy et al., 2020; Poeschel et al., 2021; Laufer et al., 2022) and CAPIO[28] (Martinelli et al., 2023), which allow the usage of streaming workflow, where I/O is executed in background when the computational part is

[25] https://casa.nrao.edu/
[26] https://wsclean.readthedocs.io/en/latest/
[27] https://adios2.readthedocs.io/en/v2.10.1/
[28] https://github.com/High-Performance-IO/capio

still running. Another approach is to compress data at the beginning of the pipeline thanks to algorithms like MGARD[29] (Gong et al., 2023; Williamson et al., 2024), which permits both lossless and lossy compression.

The code presented in this thesis is able to address many of the preceding requests mentioned in this section, since it has been fully parallelized with MPI+OpenMP, and even fully ported to GPUs with CUDA/HIP. Libraries for very specific purposes like communications and Fourier Transforms have been implemented in the GPU version, resulting in a code which loads the memory on the accelerator at the beginning, and only at the end, after all the computing steps the memory is transferred back to the host to write the final image. This result is of utmost importance, since the worst caveat in GPU computing is by far managing the memory transfers back and forth between the host and the device. This solution allows to harness the full GPU computing power, i.e. their exceptional ability in crunching floating point numbers, with less care about memory overhead. However, the code has been thought to run at the most powerful accelerated supercomputers, because to achieve the best performance it needs high speed GPU-GPU interconnects inside each computing node and the best networks available for node-node connections.

With these available technologies, we managed to achieve a speedup factor of $\sim 150$ in the largest configurations available between the best CPU and GPU configurations, both run at the Leonardo supercomputer. In a nutshell, the fastest CPU configuration takes $2$ hours and $50$ minutes, compared to the $67$ seconds taken by the fastest GPU configuration. Original data have been provided by LOFAR VLBI, precursor of what will be delivered from SKA, as anticipated above.

For the energy-to-solution, intermediate tests have been performed with smaller LOFAR HBA datasets, run at the Setonix-CPU and Setonix-GPU partitions. The results show that using GPUs one gets an energy gain factor of $\sim 12$, confirming that GPUs have a better $Flops/W$ than CPUs, and also that GPU communication libraries turn out to be much faster than standard MPI ones. GPU computing is quickly going towards being the best compromise in both computing intensive and memory-bound algorithms.

[29] `https://github.com/CODARcode/MGARD`

## 1.10 Summary

Most powerful High Performance Computing platforms are increasingly energy demanding, eventually becoming unsustainable if green computing does not come into play. Green computing encompasses hardware solutions with the newest CPU/GPU/FPGA technologies, and green algorithms as well from the software point of view. The "Holy Graal" is to write codes which adapt perfectly to the specific architectures and achieve the best performance available. However, a code performing at best on a particular platform may turn out to have a different behaviour on a different one, this is because of the hardware's difference. Code's portability is much easier to reach than performance portability. We discussed about CUDA/HIP/OpenMP solutions for GPU computing and argued that they're portable, but how much the performance changes from a machine to another has to be investigated further.

At the same time, current radio-interferometers require always larger memory and computing time, eventually reaching an entire exascale machine to process input data for SKA. This will result in several *MW* consumed, which is a huge energy requirement in this age of environmental problems and climate changes. Approaches to renewable energy sources have been adopted, like in (Geveler et al., 2017) and at the Pawsey Supercomputing Centre in Perth, where solar panels are used to feed at least partially the energy requirement of the machines. Furthermore, machines are cooled with a liquid-cooling system exploiting the water in the aquifer under Perth, but people suggest that will become an outstanding challenge to feed and cool down entire exascale platforms with renewable energy. From the software viewpoint, the development of green algorithms is unavoidable, keeping an eye to the possibility to use smaller portions of the machines without impacting much on the performance. This can be reachable when the codes are memory dominated instead of computationally dominated, as we will see in the next chapters. Considering that CPU frequency has saturated with the end of the Dennard scaling introduced in Section 1.4, parallel computing becomes the only chance if we want to do High Performance Computing.

This thesis will be structured as follows:

**Chapter 2** Dedicated to the communication problem, and how full parallelism cannot be exploited when parallel workers need to communicate, in both reduce operation and parallel Fourier transform, and how I included the GPU-GPU communication in both operations. The next part of the chapter is dedicated to the radio imaging code introduction and what has been my contribution in its development. In particular, imaging challenges in radio imaging will be stressed with the description of the fully parallelized and/or GPU offloaded pipeline with the theory of w-stacking gridder.

**Chapter 3** Dedicated to the hybrid reduce implementation which I've contributed to develop by combining MPI and OpenMP parallelization paradigms, and to the energy measures that we did on two different machines. The chapter will encom-

pass a detailed description on how to access CPU energy counters with low-level drivers and high-level libraries. A particular attention has been put in the chapter to the algorithmic imprint in energy-to-solution, to show that energy is not only an integral over time.

**Chapter 4** I will present results for our code for single-node configurations taking an eye on the I/O problem introduced in Section 1.9, run at the Leonardo machine available at CINECA[30].

**Chapter 5** I will present results for our code for both single-node and multi-node configurations run at the Leonardo machine and I will focus the attention on the full GPU implementation and on the communication bottleneck.

**Chapter 6** I will present results for our code for the energy-to-solution and time-to-solution trade-off run at the Setonix machine available at the Pawsey Supercomputing Centre[31].

**Chapter 7** Conclusions will be drawn and discussed.

[30] https://www.cineca.it/it
[31] https://pawsey.org.au/

*This page intentionally left blank.*

# 2

# The reduce problem and the RICK code

## 2.1 Introduction

The focus of this chapter is to introduce the key algorithm that has been optimized in this thesis work and its relevance in the radio imaging code that I developed with my collaborators. We already stressed many times in Chapter 1 that communication is a big overhead in true scientific codes, especially when the number of computing nodes used is large. To be more quantitative, (Gheller et al., 2023) found out that communication impact is estimated to be around $80 - 90\%$ of the total runtime. The **MPI Reduce** topic will be introduced, which is the MPI function that is needed by the imaging code to combine together the visibility data pertaining to each parallel worker, when the MPI option is active. MPI Reduce has been implemented with several algorithms, each one with a specific purpose, from which we can extract logarithmic tree, which has low latency contribution but it suits better for power-of-two number of parallel workers, and the ring, which is multi-purpose and its performance is not influenced by the number of workers, but has in the meantime a higher latency contribution. The implementation given by the MPI standard will be described in Section 2.2, while the more "exotic" topic of GPU reduce will be the subject of Section 2.3, which has been implemented on the imaging code as well. Another communication paradigm, which is not directly related to the reduce function but is however fundamental in our code, is NVSHMEM[1] and is necessary for the Fast Fourier Transform implementation available in the code, discussed in Section 2.4. After the discussion of the relevant algorithms and their relative implementations, the code itself will be introduced in Section 2.5 and the details will be the topic of Section 2.6.

[1] `https://developer.nvidia.com/nvshmem`

## 2.2 MPI Reduce

The main paradigm for distributed parallel computing is MPI, again, Message Passing Interface. In MPI several processes, i.e. MPI tasks or ranks, are spawned. The most fundamental feature is that each MPI task is assigned a unique memory portion, and it cannot access to the memory owned to the other tasks. In **shared-memory** all the parallel workers are able to access each other's memory, and this means that total memory is simply given by the available memory of the machine. With **distributed-memory** you can connect several machines together and the total memory available increases with the computing resources.
As discussed in Section 1.7, in scientific codes parallel workers usually need to exchange each others' data, meaning that MPI tasks need to communicate in some way. This is achieved by sending and receiving messages, i.e. each process allocates a buffer in which it copies the relevant quantities and sends it to another one, which in turn allocates its own buffer to receive the message. It is important to state that each message is tagged, and for a successful communication both send and receive calls must have the same tag. These MPI functions can have two communication protocols, because they can be either blocking or non-blocking. In the former case the code execution stops at the specific MPI function until the entire message is sent or received, while in the latter the code execution keeps going, until the programmer puts in a synchronization function that waits until the communication is over.
However, the actual power of MPI is given by the **collectives**, which are operations involving all the MPI tasks in a specific **communicator**. Creating a communicator means grouping a subset of processes for purpose of code's requirements or hardware's topology. The default communicator in MPI groups all the MPI ranks spawned. MPI collectives include *MPI Gather*, where all the tasks have a portion of a vector dataset and the programmer wants a target task to gather the entire vector, *MPI Scatter*, which is the opposite operation, where one MPI task scatters its vector sending a portion to all the other tasks, *MPI Broadcast*, allowing one MPI task to send a meaningful dataset to all the others, and *MPI Reduce*, the one we focused on, in which each MPI task has a partial dataset that needs to be combined to a target rank, the operation for the final combination being a summation, multiplication, finding the maximum or the minimum value among all the ranks, and so on. *MPI Gather* and *MPI Reduce* have their counterparts in which no target task is selected, and the operations become all-to-all, named *MPI Allgather* and *MPI Allreduce*, respectively. For the sake of completeness, each of these functions has its non-blocking counterpart, needing synchronization barriers exactly as in the send/receive operations.
We payed particular attention to the *MPI Reduce* where partial data are summed to the final data, because it's the function needed by the code we will present. When large arrays needs to be reduced and many MPI tasks are involved in the reduction, this function introduces a strong overhead and leads to communication-bound codes. How the *MPI Reduce* is implemented depends on the particular MPI implementation,

but usually some specific algorithmic choices are made, like tree or ring algorithms, as described in (Shan et al., 2018). The theoretical runtime of the tree algorithm (the default algorithm available in OpenMPI implementation) is estimated as:

$$\begin{aligned} T_{Reduce} &= (T_{comm} + T_{comp}) \times N \\ &= \left(\beta \frac{D}{N} + \lambda \log N\right) \times N + \left(\frac{D}{N} t_{sum}\right) \times N \\ &= (\beta + t_{sum})D + \lambda N \log N \end{aligned} \tag{2.1}$$

In Equation 2.1, $T_{comp}$ is the computation time given by:

$$T_{comp} = \frac{D}{N} t_{sum} \tag{2.2}$$

Where $t_{sum}$ is the time for a single summation, $D$ is the datasize and $\lambda$ is the communication latency time. For the new architectures, $\lambda < 0.6\mu s$, and its $N \log N$ dependency means that we would need $\sim 10^5$ MPI tasks to get a latency of $1s$. $\beta$ is a parameter that includes the bandwidth and the network topology, which are different in each HPC architecture.

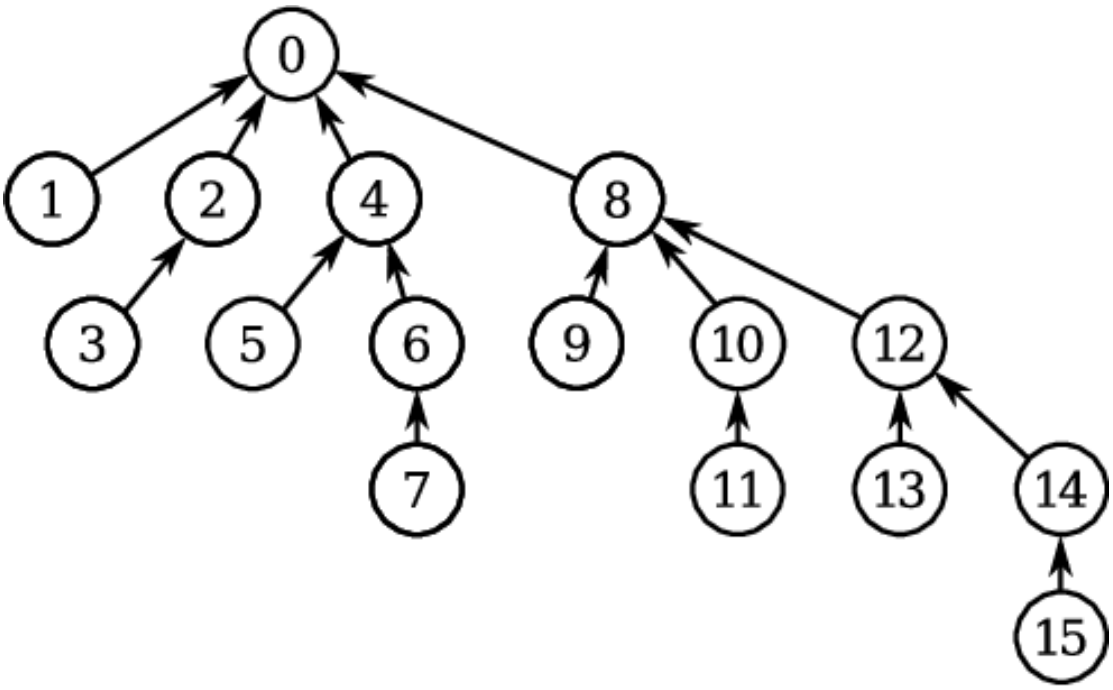



**Figure 2.1:** *Illustration of how a reduce with a logarithmic tree algorithm is implemented by MPI. Source:* `https://www.researchgate.net/figure/MPI-Reduce-binomial-tree-i-i-16_fig1_374772011`

Figure 2.1 is an illustration scheme of how a tree algorithm reduce is implemented by MPI. For construction, partial communication is performed step-by-step pairwise, eventually collecting the result to the target task. This implies that the algorithm is perfectly adapt when $N$ is a power of two, but needs to be adjusted to fit communications

between other task configurations.

For a ring algorithm, which is the one in which we put the most effort, the theoretical runtime is:

$$T_{Reduce} = (\beta + t_{sum})D + \lambda N^2 \tag{2.3}$$

The difference with Equation 2.1, is that here the $\lambda$ term increases with $N^2$, meaning that in a ring implementation one in principle can get a latency of $1s$ with $10^3$ MPI tasks. The ring algorithm has a higher latency impact but it does not introduce overheads in both performance and programmer effort when $N$ is not a power of two, which does happen for the tree algorithm.

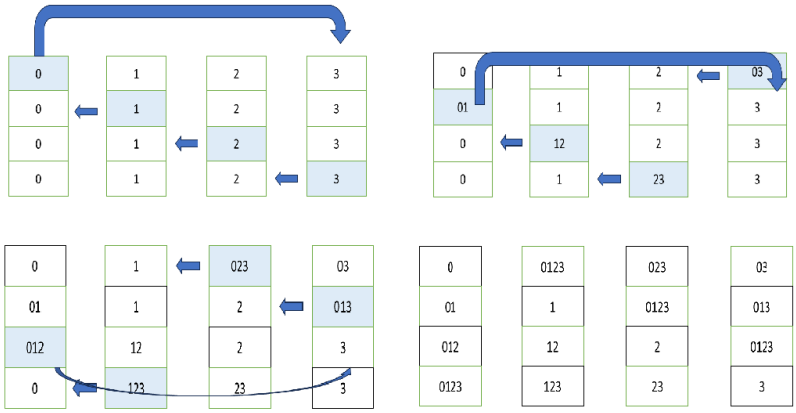



**Figure 2.2:** *Illustration of the four steps of the ring algorithm for $N = 4$. The numbers in each block represent the MPI tasks whose data have already been reduced. The shaded blocks are the communication data. Each process holds one block of the final result. A memory movement is needed to transfer data to the target rank.*

Figure 2.2 displays a scheme about how a ring reduce is implemented by MPI. The example shows the case $N = 4$, but in no extent this means that it's the best configuration. Indeed, the picture can be generalized to whatever $N$, and the algorithm doesn't lose any performance power even when $N$ is an odd number.

## 2.3 GPU reduce

In Section 1.3 and Section 1.5 we stated that GPUs had been originally thought for heavy arithmetic intensive operations, being constituted by thousands of small cores with few logic gates allowing just summation, multiplication and transcendental functions. The main bottleneck in GPU computing is however the memory transfer with the host, which in every case is unavoidable and in most cases memory copies back and forth are needed for hybrid codes, which run on both CPU and GPU. The updated libraries which have been implemented by vendors permit software developers to perform memory-bound and communications operations in a very efficient way, such that passing at each step from the CPU is not mandatory anymore.

The MPI standard does have its implementation for GPU-GPU communication[2] (H. Wang et al., 2013; Faraji et al., 2014; Faraji et al., 2018; Hanford et al., 2020; Shafie Khorassani et al., 2021; C.-C. Chen et al., 2023).

GPU aware execution model using OpenMP Offload

```
// Mark data to be copied from host to device
#pragma omp target data map(to: rank, values[0:num_values], num_values) use_device_ptr(values)
{
   // Compute on GPU
   #pragma omp target parallel for is_device_ptr(values)
   for (i = 0; i < num_values; ++i) {
       values[i] = values[i] + rank + 1;
       printf("[Within GPU:] values[%d]=%d \n",i,values[i]);
       }

    // Send device buffer to rank 0 without the use of from:values
    MPI_Send(values, num_values, MPI_INT, dest_rank, tag, MPI_COMM_WORLD);
}
```

**Figure 2.3:** *Portion of an code offloaded with OpenMP with MPI GPU-aware utilization. In particular, the values pointer is recognized as a GPU pointer, and is sent to another GPU without passing throgh the host. Source:* `https://www.intel.com/content/www/us/en/docs/oneapi/optimization-guide-gpu/2023-2/intel-mpi-for-gpu-clusters.html`.

Figure 2.3 is an example of how MPI GPU-aware is utilized in a simple OpenMP offloaded code. There's no reduce here but an MPI Send, in which *use device ptr* declares that the corresponding pointer is a GPU pointer, and MPI operates directly between two GPUs. The implementation exploits the hardware interconnection, i.e. NVLink or NVSwitch for NVIDIA[3], InfinityFabric for AMD[4], Intel Xe[5], when two or more devices are on the same node. When multiple devices are on different nodes, MPI is forced to go through the network. However, a caveat of this MPI GPU communication scheme is the lack of GPU direct operations in many collectives, i.e. MPI works perfectly fine

[2] https://developer.nvidia.com/blog/introduction-cuda-aware-mpi/
[3] https://www.nvidia.com/en-us/data-center/nvlink/
[4] https://www.amd.com/content/dam/amd/en/documents/instinct-tech-docs/other/56978.pdf
[5] https://www.intel.com/content/www/us/en/docs/oneapi/optimization-guide-gpu/2023-0/intel-iris-xe-gpu-architecture.html

for point-to-point operations, like send/receive, but is still forced to pass through the host for most collective operations, leading to a CPU transfer overhead which makes MPI GPU-aware not useful for most purposes. In fact, some MPI implementations do have this awareness for MPI Allreduce, but not for MPI Reduce yet.

To handle this limitation, vendors have developed libraries capable to perform both point-to-point and collective operations to connect directly two or more GPUs without the needing to pass from the host, thus avoiding memory copy overhead. These libraries are the NVIDIA Collective Communications Library (NCCL)[6] and the ROCm Collective Communications Library (RCCL)[7] for AMD. Again, they utilize high-speed interconnect inside the node and go through the network between nodes. Collectives have been implemented with several algorithms, for instance both binary tree and ring in Figure 2.1 and Figure 2.2 are available, with the latter being the default one when no environmental variable is set.

Figure 2.4 is a code portion which explains a simple NCCL utilization. In particular, the operation to be performed is the Allreduce, but the scheme is exactly the same for the Reduce. At the beginning, NCCL is attached the MPI communicator, which is necessary when one process per GPU is assigned. The syntax is the same as the MPI one. In this case, both the send and receiving buffers are already distributed amnong the GPUs, ready to be summed up together for the reduce and then broadcast to conclude the operation. RCCL implementation is obtained by trivially replacing CUDA calls with HIP calls. NCCL collects together GPUs in the same node to perform an initial partial reduce (the algorithm can be changed but is set to be a ring by default), and then these partial sums are sent to the master process which is in charge to communicate and sum its data with the ones coming from all the others (A. Li et al., 2019; Nvidia, n.d.; NVIDIA, 2017; He et al., 2024).

I have implemented NCCL and RCCL reduce in the radio imaging code introduced in Section 2.5 with the purpose to accelerate the communication and to have a full GPU version, since leaving the communication on the CPU would unavoidably lead to memory transfers back and forth between host and device. The accelerated reduce permits to hold the entire dataset (in this case the grid) during the whole gridding operation.

[6] `https://developer.nvidia.com/nccl`
[7] `https://rocm.docs.amd.com/projects/rccl/en/latest/`

```
  //picking a GPU based on localRank, allocate device buffers
  CUDACHECK(cudaSetDevice(localRank));
  CUDACHECK(cudaMalloc(&sendbuff, size * sizeof(float)));
  CUDACHECK(cudaMalloc(&recvbuff, size * sizeof(float)));
  CUDACHECK(cudaStreamCreate(&s));


  //initializing NCCL
  NCCLCHECK(ncclCommInitRank(&comm, nRanks, id, myRank));


  //communicating using NCCL
  NCCLCHECK(ncclAllReduce((const void*)sendbuff, (void*)recvbuff, size, ncclFloat, ncclSum,
        comm, s));


  //completing NCCL operation by synchronizing on the CUDA stream
  CUDACHECK(cudaStreamSynchronize(s));


  //free device buffers
  CUDACHECK(cudaFree(sendbuff));
  CUDACHECK(cudaFree(recvbuff));


  //finalizing NCCL
  ncclCommDestroy(comm);


  //finalizing MPI
  MPICHECK(MPI_Finalize());


  printf("[MPI Rank %d] Success \n", myRank);
  return 0;
}
```

**Figure 2.4:** *Portion of an code offloaded with CUDA with NCCL utilization for the Allreduce operation. In this case NCCL is attached to the standard MPI communicator and one process per GPU is assigned. Source:* `https://docs.nvidia.com/deeplearning/nccl/user-guide/docs/examples.html`.

## 2.4 NVSHMEM and cuFFTMP

Aside NCCL, which explicitly uses MPI point-to-point and collective operations, there is another communication paradigm for GPU direct, called NVSHMEM (NVIDIA OpenSHMEM), which is thought to allow one-sided communications, i.e. information exchange not involving a send and a receive directly (Potluri et al., 2015; Hsu et al., 2020; Hsu et al., 2021). There is one process which communicates with another one by creating a "shared-memory channel", in which the latter can "access" the memory of the former without any buffer and any message exchange. This communication scheme enables what is called RDMA (Remote Direct Memory Access) (Liu et al., 2003). In Direct Memory Access, a *shared window* is created to get a channel between two or more processes, by using a shared-memory approach even among MPI tasks. A direct exploitation of these windows will be discussed in Chapter 3 when I will introduce our customized MPI Reduce implementation. Utilizing MPI calls, this operation is possible since sender process "puts" the relevant data in the window (which needs to be created and allocated on its own) with an *MPI Put*, while the receiver one "gets" it with an *MPI Get*. An example of standard MPI with asynchronous send/receive operations and NVSHMEM is shown in Figure 2.5.

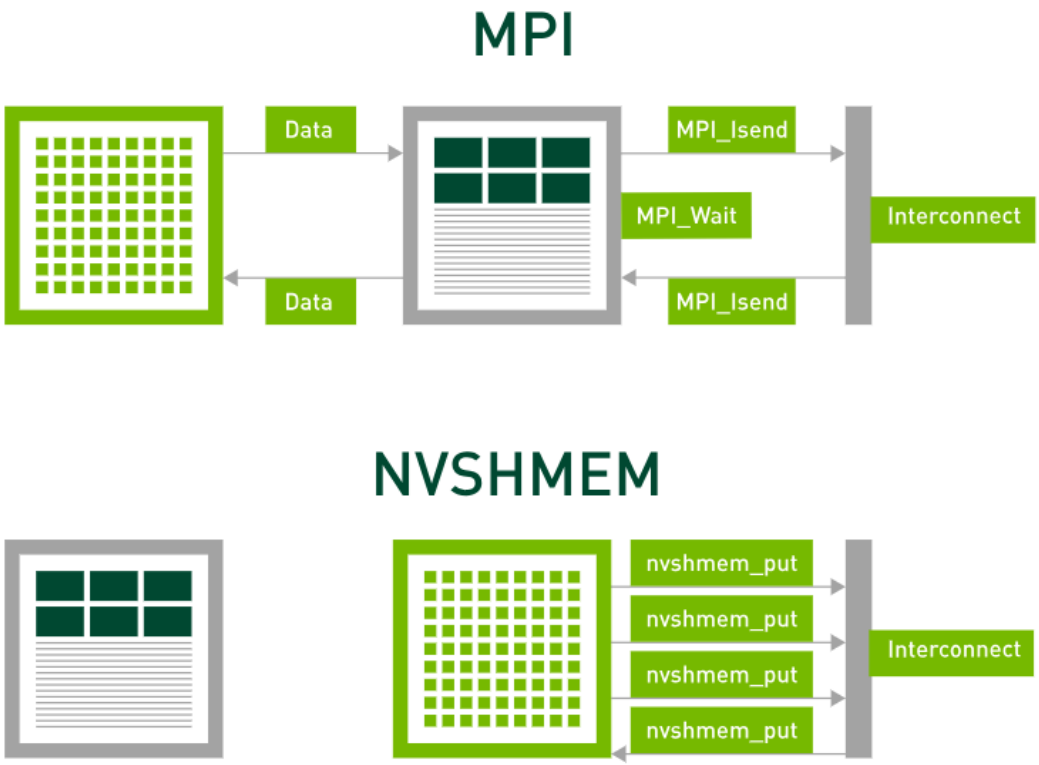



**Figure 2.5:** *Existing communication models, such as Message-Passing Interface (MPI), orchestrate data transfers using the CPU. In contrast, NVSHMEM uses asynchronous, GPU-initiated data transfers, eliminating synchronization overheads between the CPU and the GPU. Source:* `https://developer.nvidia.com/nvshmem`.

```
#include <stdio.h>
#include <cuda.h>
#include <nvshmem.h>
#include <nvshmemx.h>

__global__ void simple_shift(int *destination) {
      int mype = nvshmem_my_pe();
      int npes = nvshmem_n_pes();
      int peer = (mype + 1) % npes;

      nvshmem_int_p(destination, mype, peer);
}

int main(void) {
    int mype_node, msg;
    cudaStream_t stream;

    nvshmem_init();
    mype_node = nvshmemx_my_pe(NVSHMEMX_TEAM_NODE);
    cudaSetDevice(mype_node);
    cudaStreamCreate(&stream);

    int *destination = (int *) nvshmem_malloc(sizeof(int));

    simple_shift<<<1, 1, 0, stream>>>(destination);
    nvshmemx_barrier_all_on_stream(stream);
    cudaMemcpyAsync(&msg, destination, sizeof(int), cudaMemcpyDeviceToHost, stream);

    cudaStreamSynchronize(stream);
    printf("%d: received message %d\n", nvshmem_my_pe(), msg);

    nvshmem_free(destination);
    nvshmem_finalize();
    return 0;
}
```

**Figure 2.6:** *Example of a ring communication scheme in an NVSHMEM code, for simplicity MPI is not explicitly used in this example. Source:* `https://docs.nvidia.com/nvshmem/archives/nvshmem-101/developer-guide/index.html`.

Figure 2.6 shows a ring communication scheme with NVSHMEM in which each GPU communicates directly with its nearest neighbour in the same "team". This code does not use MPI, thus is a simplified scheme, with MPI NVSHMEM is attached to the MPI communicator and each process is attached to a GPU. However, the MPI extension works with the same idea shown in the example. I have not directly used NVSHMEM in the radio imaging code, but it's fundamental for one particular library that the code needs, which is the cuFFTMP (CUDA Fast Fourier Transform for Multi-Process)[8] (Das, n.d.; Cambier et al., 2022; Ayala et al., 2022; Verma et al., 2023).

Figure 2.7 schematically shows how to write a 3D distributed Fast Fourier Transform on GPUs, by attaching the MPI communicator to the cuFFTMP, which utilizes NVSHMEM under the hood, and then the allocation of descriptors, i.e. C++ constructs storing all the information about the array to be transformed. It is important to add that after the descriptor execution, which contains the actual calculation, a device to device communication is usually needed to put the data in the original order, otherwise the output will display shuffled results.

[8] `https://docs.nvidia.com/hpc-sdk/cufftmp/index.html`

In the end, the radio imaging code utilizes NCCL for the reduce and NVSHMEM for the FFT. To avoid conflicts in communication paradigms, the environmental variable $NVSHMEM\ DISABLE\ NCCL = 1$ must be set, since when both libraries are linked in the same code, NVSHMEM is by default forced to use an MPI-like approach as NCCL, which leads to bugs and performance degradation.

```cpp
#include <cufftMp.h>
#include <mpi.h>
#include <vector>
#include <complex>

int main() {

    // Initialize MPI, pick a device
    MPI_Init(NULL, NULL);
    MPI_Comm comm = MPI_COMM_WORLD;

    int rank, size;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    int ndevices;
    cudaGetDeviceCount(&ndevices);
    cudaSetDevice(rank % ndevices);

    // Allocate CPU memory
    size_t N = 32;
    std::vector<std::complex<float>> cpu_data((N / size) * N * N, {314, 0});

    // Create plan, attach to communicator, make plan
    cufftHandle plan = 0;
    size_t workspace;
    cufftCreate(&plan);
    cufftMpAttachComm(plan, CUFFT_COMM_MPI, &comm);
    cufftMakePlan3d(plan, N, N, N, CUFFT_C2C, &workspace);

    // Allocate memory, copy CPU data to GPU
    cudaLibXtDesc *desc;
    cufftXtMalloc(plan, &desc, CUFFT_XT_FORMAT_INPLACE);
    cufftXtMemcpy(plan, desc, cpu_data.data(), CUFFT_COPY_HOST_TO_DEVICE);

    // Run C2C FFT Forward
    cufftXtExecDescriptor(plan, desc, desc, CUFFT_FORWARD);

    // Copy back to CPU
    cufftXtMemcpy(plan, cpu_data.data(), desc, CUFFT_COPY_DEVICE_TO_HOST);

    // Data in cpu_data is now distributed along the Y dimension, of size N * (N / size) * N
    // Test by comparing the very first entry on rank 0
    if(rank == 0) {
        if(cpu_data[0].real() == 314 * N * N * N) {
            printf("PASSED\n");
        } else {
            printf("FAILED\n");
        }
    }

    // Cleanup
    cufftXtFree(desc);
    cufftDestroy(plan);
    MPI_Finalize();

}
```

**Figure 2.7:** *Scheme of a complex-to-complex distributed 3D Fast Fourier Transform on GPUs. Descriptors are C++ constructs needed to store all the information about the array to be transformed. Source:* `https://docs.nvidia.com/hpc-sdk/cufftmp/getting_started/index.html`.

## 2.5 Introduction of the RICK code

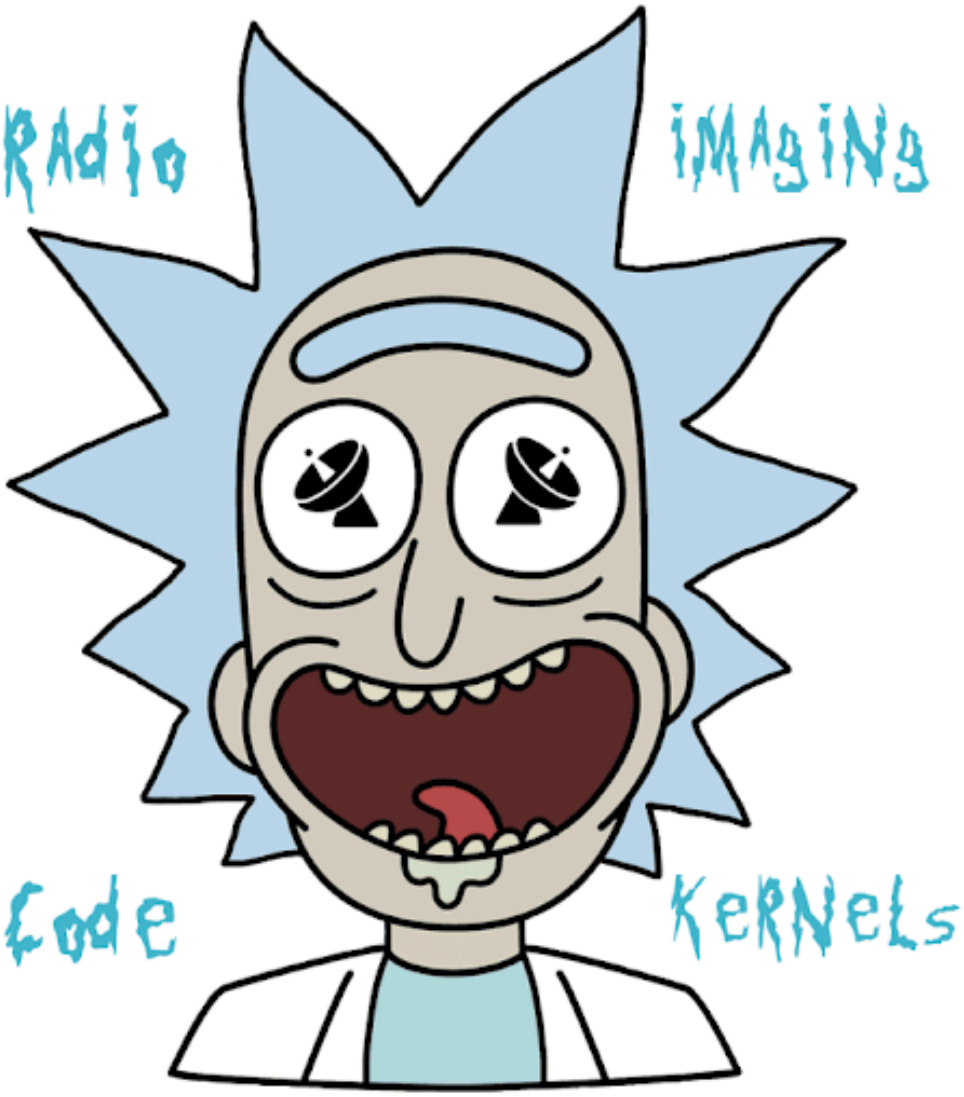



**Figure 2.8:** *Code mascot's from the popular Rick and Morty animated series. Credit: Emanuele De Rubeis.*

Radio astronomy is currently witnessing a rapid increase in the volume of data that are being collected by radio-interferometers like the LOw Frequency ARray (LOFAR, van Haarlem et al., 2013), MeerKAT (Jonas et al., 2016), the Murchison Widefield Array (MWA, Mitchell et al., 2010), the Australian Square Kilometre Array Pathfinder (ASKAP, Johnston et al., 2007). These instruments can produce petabytes of data every year, precursors of what will be delivered by the Square Kilometre Array (SKA[9]), expected to generate hundreds of petabytes of data each year. In (Gheller et al., 2023) we (with the collaborators in my team) have introduced a novel approach for implementing the $w$-stacking algorithm (A. R. Offringa et al., 2014) for imaging on state-of-

[9] `https://www.skatelescope.org/`

the-art High Performance Computing (HPC) systems, effectively exploiting heterogeneous architectures, consisting of thousands of multi-core CPUs equipped with accelerators like GPUs to enhance computational performance while minimising power consumption.

Imaging is a computationally intensive step in the data processing pipeline (T. Cornwell, n.d.), requiring a significant amount of memory and computing time. This is due to operations such as gridding, which involves resampling the observed data on a computational mesh, and fast Fourier transform (FFT), which converts between Fourier and real space. The computational demands increase when dealing with observations that have large fields of view, especially in current radio interferometers and at low frequencies. This is because curvature effects cannot be ignored, making the problem fully three-dimensional. In such cases, the "$w$-term" correction needs to be introduced (see Section 2.6, T. J. Cornwell et al., 2008; A. R. Offringa et al., 2014).

The gridding, FFT, and $w$-correction steps are integrated into the so-called *$w$-stacking gridder* algorithm. These steps are suitable to distributed memory parallelism, exploiting parallel FFT solutions and relying on a Cartesian 3D computational mesh, that can be effectively distributed and efficiently managed across different processing units, resulting in good scalability on large HPC architectures. In modern HPC systems, besides distributed processing, performance can be achieved through multi-core and accelerated (many-core) computing based on GPUs. Current trends suggest that some form of heterogeneous computing will be prevalent in emerging architectures (e.g., Keckler et al., 2011). Therefore, the ability to fully exploit new heterogeneous and many-core solutions is of paramount importance towards achieving optimal performance. In (Gheller et al., 2023) we have described the enabling of the gridding and $w$-correction algorithms to multi/many-core parallelism. We have demonstrated how utilising GPUs can significantly decrease the time to solution thanks to their outstanding performance. In addition, we have pointed out how multi/many-threads solutions allow using a smaller number of Message Passing Interface (MPI) tasks compared to a pure MPI set-up, mitigating communication-related problems. However, to fully harness power of GPU acceleration, it is critical to enable the full code for GPUs, including MPI communication. This is essential to:

- speed-up all algorithmic components, in particular the FFT step. The performance of the code is hindered by non-accelerated parts, which become bottlenecks;
- avoid unnecessary data movements between the host and the device, required, in the code presented in (Gheller et al., 2023), to implement data communication among CPUs and for the FFT transform. Efficient code performance relies heavily on minimising data movements;
- exploit high performance interconnect available among GPUs on the same computing node.

In this thesis, we discuss how we have tackled these challenges by exploiting com-

pilers and libraries provided by different vendors, i.e. NVIDIA HPC Software Development Kit (SDK[10]). These libraries enable us to perform accelerated FFT calculations and to optimise inter-GPU MPI communication. This leads to a code that, once loaded the input visibilities from the file system, can fully run on GPUs; the CPUs are used once more only at the end of the calculation, solely for the purpose of saving the final image to a file. In addition, we present a solution based on FFTW that allows the utilisation of hybrid multi-core plus MPI calculation of the Fourier transform on CPUs. This is crucial to reduce the communication between MPI tasks while also providing an efficient and portable solution that is not dependent on the NVIDIA SDK.

The code, called Radio Imaging Code Kernel (RICK[11]) (a friendly mascot is shown in Figure 2.8), is developed using the C programming language standard (with extensions to C++ only to support GPUs through CUDA). Alongside CUDA, in (Gheller et al., 2023) we have introduced the Open Multi-Processing (OpenMP) support for offloading gridding and w-stacking operations to accelerators, providing a portable solution even for GPU implementation. The code is compatible with various computing platforms, although optimal performance requires the availability of suitable hardware and software solutions.

Parallelism has been exploited supporting multi-core processors (via OpenMP) and distributed HPC architectures (via MPI). Throughout the thesis, we refer to *computing* or *processing unit* as the computing entity addressing some parts of the work. In the case of parallel work based on MPI, a computing unit is a single core (mapping to an *MPI task*). In the case of multi-threaded OpenMP implementation, it is a multi-core CPU. In the case of accelerated computing, it is a GPU.

[10] `https://docs.nvidia.com/hpc-sdk/index.html`
[11] The code is publicly available here: `https://github.com/ICSC-Spoke3/RICK`

## 2.6 The $w$-stacking gridder

An interferometer measures complex visibilities $V$ related to the sky brightness distribution $I$ as:

$$V(u,v,w) = \int\int \frac{I(l,m)}{\sqrt{1-l^2-m^2}} \times e^{-2\pi i\left(ul+vm+w(\sqrt{1-l^2-m^2}-1)\right)} dl\,dm \tag{2.4}$$

Where $l, m$ are the sky coordinates while $u, v, w$ are the coordinate of the baselines in units of $\lambda$, where $w$ is chosen to be in the direction of the source. For a particular interferometer, the so-called $u, v$ plane coverage is the distribution of the baselines in $\lambda$ units as seen from the source at infinity: each projected baseline corresponds to a point of coordinates $u, v$ in the Fourier space. For small fields of view, the term $\sqrt{1-l^2-m^2}$ is close to one, and Equation 2.4 is an ordinary two-dimensional Fourier transform, which, in order to speed-up the computation, is solved by using a FFT-based approach. This, however, requires mapping visibilities, that are point like data, to a regular mesh that discretizes the $(u,v)$ space. This is accomplished by convolving the visibility data with a finite-size kernel, which converts it to a continuous function, which can then be FFT transformed.

When large Fields of View (FoV) are observed at once, visibility data from non-coplanar interferometric radio telescopes cannot be accurately imaged with a two-dimensional Fourier transform and the imaging algorithm needs to account for the $w$-term, which describes the deviation of the array from a plane. A possible approach to account for the $w$-term is represented by the $w$-stacking method (A. R. Offringa et al., 2014), in which the computational mesh has a third dimension in the $w$ direction and visibilities are mapped to the closest $w$-plane. Once gridding is completed, each $w$-plane is Fourier transformed separately, and a correction is applied as:

$$\frac{I(l,m)\,(w_{\max}-w_{\min})}{\sqrt{1-l^2-m^2}} = \int_{w_{\min}}^{w_{\max}} e^{2\pi i w(\sqrt{1-l^2-m^2}-1)} \times \iint V(u,v,w) e^{2\pi i(ul+vm)} du\,dv\,dw \tag{2.5}$$

Overall, this algorithm is faster than $w$-projection (T. J. Cornwell et al., 2008) thanks to its algorithmic characterisation, especially when the gridding is the dominating cost of the algorithm (see Tab. 1 in A. R. Offringa et al., 2014) and results in slightly lower imaging errors, making it commonly used for wide-field radio interferometers such as LOFAR and, in the near future, SKA.

The $w$-stacking algorithm has been implemented in (Gheller et al., 2023) inside the $w$-stacking gridder, which I briefly recall hereafter. Two main data structures characterise the algorithm. The first is an unstructured dataset storing the $(u,v,w)$ coordi-

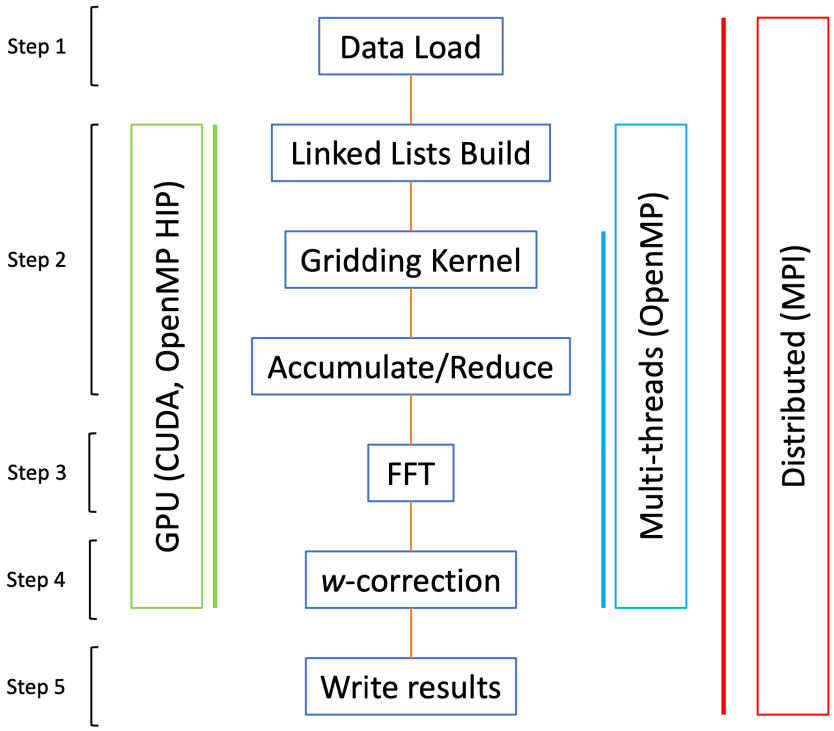



**Figure 2.9:** *Schematic code architecture and workflow of RICK, based on the one in (Gheller et al., 2023) with the new steps that we ported on GPUs (reduce and FFT). Different kind of HPC enabling are highlighted with different colours.*

nates of the antennas array baselines at each measurement time. Each baseline has a number of associated visibilities, depending on the frequency bandwidth, the frequency resolution, the number of correlations and the total observing time. A further quantity called *weight* is assigned by the correlator to each measurement: this number allows the user to adjust the balance between angular resolution, sidelobes of the beam and sensitivity (Briggs, 1995).

Visibility data are distributed among multiple memories in time slices of equal length. The second relevant data structure is a Cartesian computational mesh of size $N_u \times N_v \times N_w$, where $N_u$, $N_v$ and $N_w$ are the number of cells in the three coordinate directions. The convolved visibilities and their FFT transformed counterpart are calculated on this mesh. The data defined on the Cartesian computational grid is partitioned among multiple tasks adopting a rectangular slab-like decomposition, assigning to each task a rectangular region of $N_{\rm mesh} = N_u \times N_w \times (N_v/N_{\rm pu})$ cells, where $N_{\rm pu}$ is the number of physical memories (MPI tasks for CPUs or the number of GPUs) adopted in the computation. The two data structures determine the algorithm's memory request.

The code consists of five main algorithmic components (shown in Figure 2.9), each supporting different types of HPC implementations. The first component reads the data in parallel and distributes equal chunks to all the MPI tasks. If GPUs are used, the data offload from the host to the device memory completes this step. The following

step performs the gridding of the visibilities. Gridding is done in successive rectangular slabs along the $v$-axis. This procedure consists of three sub-steps. In the first sub-step, an array is created for each slab, concatenating the data with $u$-$v$ coordinates inside the corresponding slab. The second sub-step is represented by the convolution with the grinding kernel:

$$\tilde{V}(u_i, v_j, w_k) = \sum_{m \in \text{measures}} V_m G((u_m, v_m, w_m), (u_i, v_j, w_k)) \tag{2.6}$$

Where $m$ is the $m$-th measurement, $(u_m, v_m, w_m)$ are its coordinates, $(u_i, v_j, w_k)$ is a computational grid point, $V_m$ is the measured visibility and $\tilde{V}$ is the visibility convolved on the mesh. For the tests presented in this thesis, the $G$ kernel is either a Kaiser-Bessel function (Jackson et al., 1991) or a Gaussian. Once boundary data among different slabs are properly managed, this part of the computation is completely local and can be executed on multiple cores or accelerators. This step is finalised by the exchange of information among different computing units, necessary to accomplish the calculation of gridded visibilities on each slab. This is a critical operation, whose detailed description is given in (Gheller et al., 2023).

The third algorithmic component performs the FFT of the gridded data, producing the real space image. Next, we apply the phase shift and reduce the $w$-planes to obtain the final image (Step 4). The fifth and final step writes the final images exploiting parallel I/O.

Gridding here involves also the communication, done thanks to the reduce operation. I've been involved in the implementation of an hybrid MPI+OpenMP reduce, subject of the next chapter, and I successfully ported it in RICK. To complete the hybrid CPU code, I adapted the original FFT, which was parallelized with MPI only[12], to work also with OpenMP[13], in order to have at the end a fully parallelized code which works with MPI, OpenMP and a combination of both. I anticipate here that the hybrid version is the best trade-off between the pure performance needed in the most arithmetic intensive parts, i.e. kernel convolution in gridding and $w$-correction, and the communication-bound ones, i.e. reduce operations and FFTs. Lowering the number of MPI tasks but compensating with OpenMP threads reduces the communication surface without in principle losing any performance, since all the available CPU cores are occupied and fully working. However, I anticipate that the arithmetic intensive part of gridding (hereafter only gridding, to separate computation from the communication I will simply say *gridding* and *communication/reduce*) and $w$-correction show a performance difference depending on the use of the hybrid or pure MPI implementation, whose intensity changes with the machine architecture.

I've contributed to the code's GPU implementation in gridding, communication and FFT. For the gridding, my work has been to optimize the initial memory transfer between host and device, thanks to the inclusion of CUDA asynchronous memory

[12] `https://fftw.org/doc/Distributed_002dmemory-FFTW-with-MPI.html`
[13] https://fftw.org/doc/Combining-MPI-and-Threads.html

copies and *streams*, which allow us to split the GPU memory loads and kernels in distinct and overlapping workflows. I am now developing a combined approach with OpenMP and streams in order to speedup the CPU-GPU communication by exploiting CPU parallelism. Of course this is possible only when single streams don't move large data, since otherwise bandwidth will soon be fulled and communications will be treated sequentially. Workflows can be either dependent or independent, and in the former case stream synchronization is mandatory. As claimed in the previous sections, I've implemented the NCCL reduce in the code by using two different streams: one for the gridding kernel and the other for the communication. Of course, the communication cannot start until all the visibilities have been gridded, such that CUDA streams are essential because we will see that the two parts are inherently entangled. I've contibuted to implement the distributed cuFFTMP on GPUs, which becomes a difficult task when the grid has already been distributed on GPUs. This indeed implies that pointers after the reduce match with FFT's descriptors intrinsic pointer alignments, and this led to copy the memory buffer to another which matches the correct memory location.

As claimed in Section 1.9, there are other libraries already performing radio imaging. We decided to compare our outputs with the WSClean ones to see if the images match. We couldn't go further since currently our code has a different normalization scheme and the final fluxes are of course different, meaning that the outputs are not binary equal. So we show the reconstructed images with the two codes in their own normalization scheme.

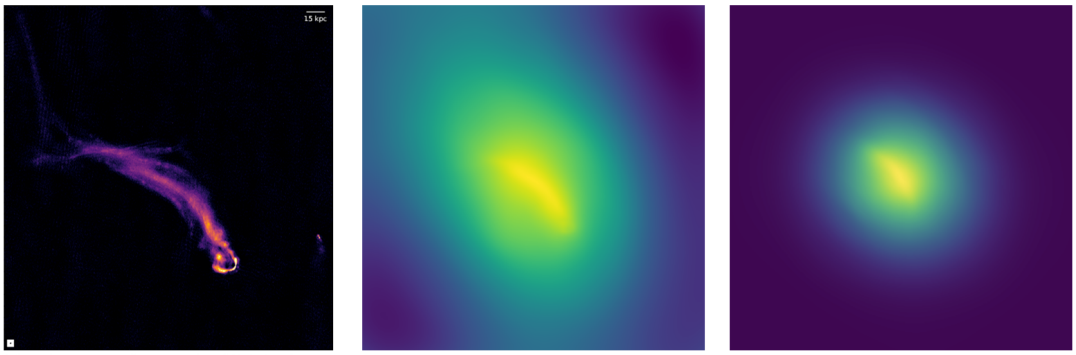


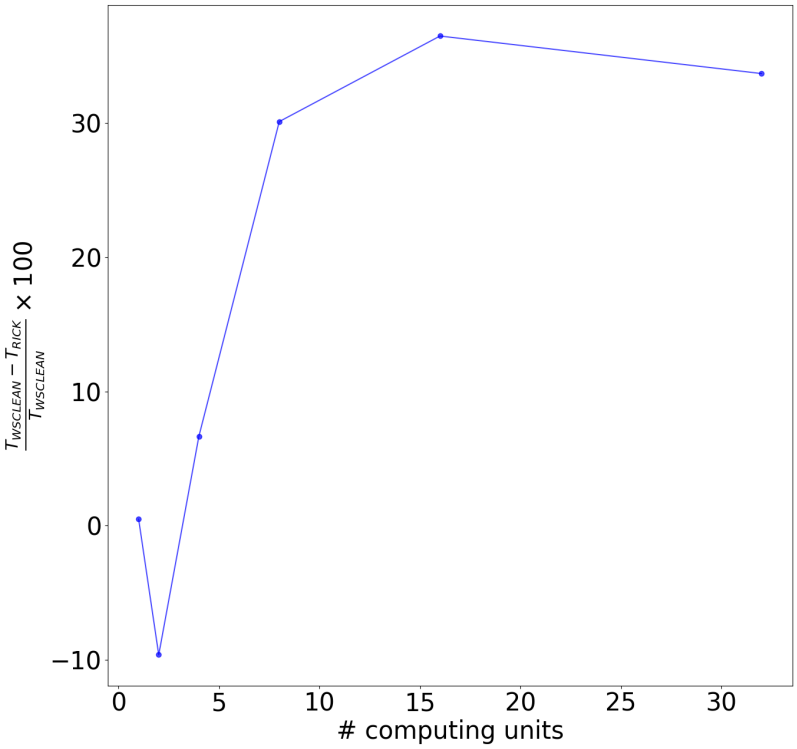

**Figure 2.10:** *Deconvolved image, WSClean image with natural weighting, RICK image. The dataset refers to the "Original TRG", a head-tail radio galaxy in the galaxy cluster Abell 2255. Credit: Emanuele De Rubeis.*

Figure 2.10 displays in the left panel the deconvolved image of a head-tail radio galaxy in the cluster Abell 2255, which is the closest one to the real object, while in the central panel and right panel there are the "dirty" images from WSClean and RICK, respectively. The weighting for the visibilities is the same, but the difference is that WSClean discerns the polarization beams and compute fluxes for just one polarization,

while RICK doesn't have this feature yet and combines together all the polarizations, performing more computation in terms of $Flops$ but at the same time losing information about sources around the central object. In the three images the normalization is different so it is out of scope checking whether the outputs are binary equal, this is instead a first hint to see that RICK is actually reproducing the right radio source. The galaxy tail is clearly recognizable in both the dirty images. For the sake of completeness, we see that the deconvolved image shows just one source, but this is due to the fact that distance scale is different between the first and the other two images.

Figure 2.11 is a quantitative comparison between RICK and WSClean, inside one computing node since WSClean has no full MPI implementation, and we had to go for OpenMP version, with up to 32 threads on the Marconi100 machine available at CINECA. RICK has been run with MPI instead, and in the upper panel we show the relative difference among the two codes, with the same number of computing units to produce the same dirty image. While in the serial case the two codes' performances overlap, and with two computing units WSClean is actually faster than RICK, by increasing $N$ the performance of RICK increases more rapidly, eventually reaching a more than $30\%$ performance gain compared to WSClean. The lower panel shows efficiencies, defined in Equation 1.5, with WSClean in green and RICK in orange. Although RICK needs MPI collectives, the efficiency is better than WSClean, whose computing units are actually working in shared-memory. However, while MPI collectives are a problem and this will be extensively discussed throughout all the thesis, the reason for WSClean poor performance is that multi-threading alone leads quickly to memory contention, since the memory bandwidth is eventually saturated by the increasing number of threads trying to access DRAM, leading to the WSClean efficiency drop. Furthermore, the full MPI implementation gives the possibility to run over multiple nodes and to avoid to split original data to fit inside one node's memory, which is actually the case in WSClean. We will see through the thesis that the best configuration which reduces impact on both memory contention and MPI collectives is the hybrid MPI+OpenMP implementation, whose results on both MPI collectives and RICK will be extensively discussed in the next chapters.

Figure 2.12 shows a more fair comparison starting from the results in the upper panel of Figure 2.11, which is represented by the blue line. The red curve is the relative difference between WSClean and RICK when the communication time in RICK is excluded from the total runtime. This is because the MPI implementation requires communications, but the WSClean OpenMP parallelization is in shared-memory, and does not include any communication overhead. In this case we see that pure parallel computation in RICK reaches up to $80\%$ better performance than WSClean.



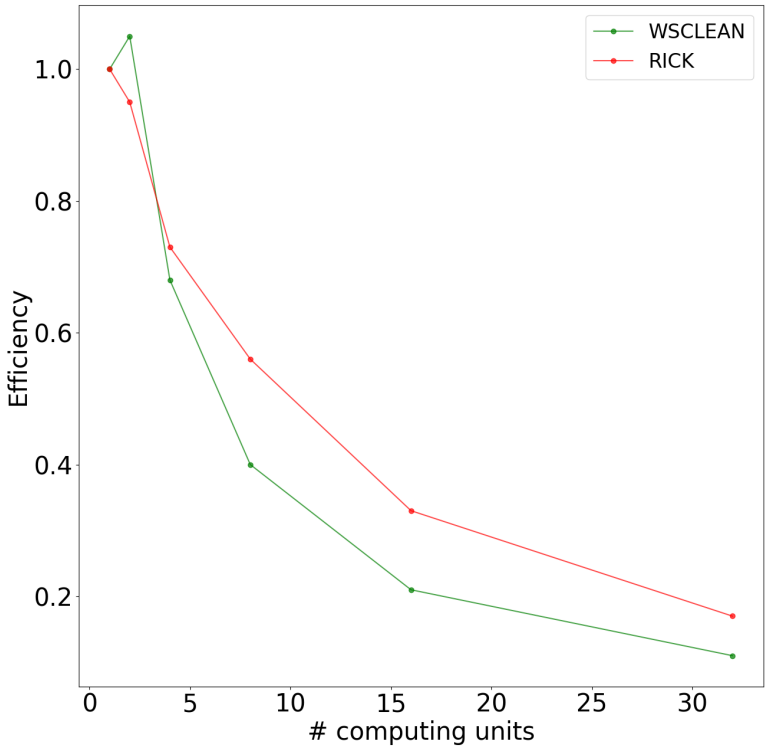




**Figure 2.11:** *Upper panel: Relative difference between WSClean and RICK codes in one node. The former has only the OpenMP parallelization paradigm, while RICK has been run with MPI. Lower panel: Corresponding efficiency, defined in Equation 1.5, of the two codes.*

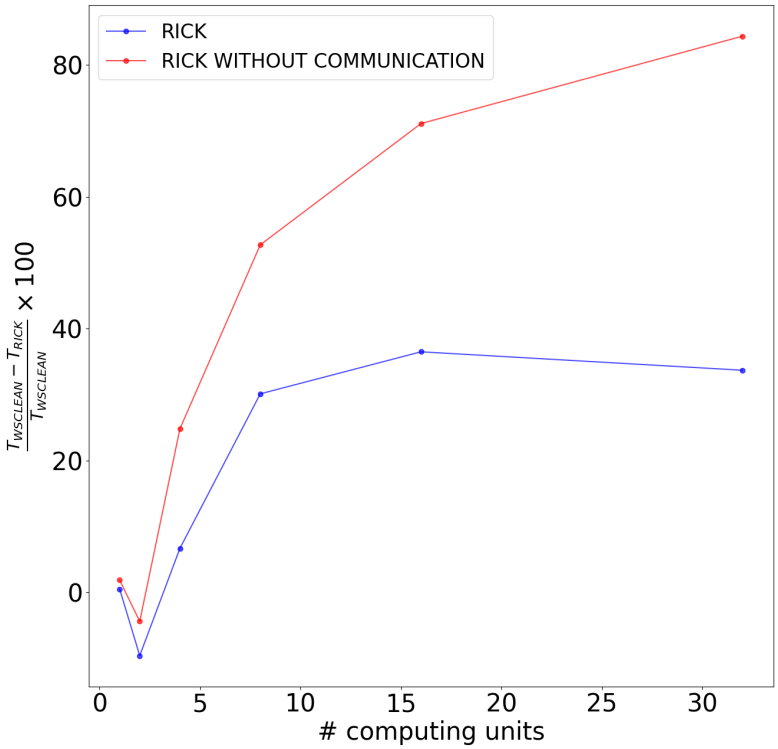



**Figure 2.12:** *Relative difference between WSClean and RICK codes in one node, with and without communication runtime included in RICK. The former has only the OpenMP parallelization paradigm, while RICK has been run with MPI.*

*This page intentionally left blank.*

# 3

# The algorithmic imprint in energy efficiency

## 3.1 Introduction

Currently the energy consumption of HPC facilities has been becoming an issue of pivotal importance and to handle with this it is necessary to mix the most energy saving architectures and the fastest and instruction-level cheapest algorithms. In particular, when many computing resources are used, MPI collectives become bottlenecks because of the not well balanced superposition between computation and communication, with this one dominating the runtime.

In this chapter we introduce an alternative implementation of the standard reduce in shared memory. In this implementation a new communicator is created, which recognizes the NUMA architecture of the machine and ranks the MPI tasks in order to achieve the maximum bandwidth, by filling all the NUMA regions in a round-robin fashion. The Reduce exploits the ring algorithm, where communications are performed among nearest neighbors. It is known that the ring has a higher bandwidth compared to the binomial tree and almost the same one of the reduce-scatter algorithm, but with the advantage that it does not need the number of MPI tasks to be power-of-two to achieve the best performance.

My collaborators and I studied the energy-to-solution and the time-to-solution of our implementation compared to the standard MPI Reduce and MPI Ireduce available in two different MPI implementations. We expect faster algorithms to be cheaper, but our aim is to investigate whether there is an algorithmic imprint in the energy consumption, i.e. whether different CPU instructions can lead to different consumption. The test machines are an Intel node with two Intel(R) Xeon(R) Gold 6126 CPU @ 2.60GHz of 12 cores each and an AMD node with two AMD EPYC 7H12 64-Core Processor @ 2.60 GHz of 64 cores each, available at the ORFEO cluster at the Area Science Park in Trieste. The CPU energy counters are different in each architecture. However, we will mainly discuss results on Intel architectures, leaving the last sections of the chapter on the discussion of the AMD results. Section 3.2 will be devoted

to the description of our hybrid reduce implementation, focusing on technical details. In Section 3.3 I will introduce Intel and AMD architectures, showing their difference in hardware structure and energy counters, with particular attention on how we have accessed these counters.

## 3.2 The hybrid reduce implementation

Pure MPI codes introduce significant overhead, especially when multiple collective calls are necessary. To reduce the communication impact, one possibility is to diminish the number or MPI tasks involved, for two reasons:

1. When each MPI task has a portion of the data and needs to communicate with all the other ranks, leading to an all-to-all communication. With less MPI tasks, there are less data sectors and less reduce calls are necessary.
2. Diminishing the number of processes means that less ranks are involved in each reduce operation, and this increases the $T_{comp}/T_{comm}$ ratio, making the code more compute-bound.

However, using less MPI tasks than available cores means that CPU is underutilized. This leads to lose in computing power and, in some MPI implementations, to errors in memory buffer allocations. To solve this issue, full parallelization is turned on with OpenMP, to saturate all the available CPU cores. This helps to speed-up the summation, because more cores are involved in the computation, and to multiple instruction parallelism, with different OpenMP threads executing different instructions at the same time.
We achieved this goal with the implementation of a **hybrid ring reduce**, which exploits both MPI and OpenMP parallelization, and it's meant to be portable, by using MPI calls available in every MPI implementation. The reduce is ring-like, with each rank communicating with its nearest-neighbours in the communicator. In each computing node MPI ranks are assigned to each NUMA region (**Non-Uniform Memory Access**), and then grouped in a intra-node communicator. Parallel workers are in **Uniform Memory Access** (UMA) when each one can access the same memory bank in the same time, whereas two parallel workers which take different times to access a specific memory region are in Non-Uniform Memory Access. NUMA is useful in order to assign specific memory banks to groups of processes, to reduce as much as possible memory contention and bandwidth saturation. MPI assigns tasks to cores in different NUMA regions in a round-robin fashion. In Intel nodes, for instance, a single socket (CPU) is a single NUMA region, so in multi-socket nodes MPI fills the two sockets by putting task 0 in socket 0, task 1 in socket 1 and so on and so forth, eventually filling all the cores available in all sockets. When all the MPI ranks are assigned (remember that for hybrid codes one is not saturating the entire machine with MPI) according to the different NUMA regions, they are grouped by the intra-node communicator in order to perform a communication only among tasks in the same node. This is needed to

force the processes to communicate with siblings in the same communicator, i.e. in the same node. The idea is to avoid as much as possible the communication through the network, which is much slower than intra-node communication, even if the processes are spread in different NUMA regions. For example, imagine to have four MPI tasks, spread over four different nodes. A collective communication needs to pass through the network at each stage several times, but if you have $2$ tasks in one node and $2$ in another node, you essentially halve the network transfers. Now, consider that the first two ranks in the first node are forced to communicate only with each other, and the same for the other ones. Both the nodes will contain a partial result which needs to be exchanged between the two nodes to obtain the final result. In one task in each node is the master, it contains the node's partial result and communicates it with the master in the other node, with just one network transfer. This is what we have done with the reduce operation.
Two communicators are created:

1. Intra-node communicator: its purpose is to group all the ranks in the same node, i.e. if $P$ is the number of MPI tasks per node, processes in each node are then ranked from $0$ to $P-1$. Rank $0$ is the node's master.
2. Inter-node communicator: its purpose is to group all the master ranks of each node, and the network communication is performed as it were just one task per node. Each master owns the node's partial result.

We stressed that in MPI each process can directly access its own memory region, while to exchange information with the other processes needs communications. However, there is the possibility to put MPI tasks in shared-memory with the **MPI shared windows**. Windows create channels of Remote Direct Memory Access (RDMA) among processes. In the intra-node communicator, ranks are in shared-memory and no send/receive operations with buffer allocations are needed. Because tasks have access to the same memory region, atomic operations are necessary for synchronization and to avoid data races, i.e. when two or more parallel workers update the same variable at the same time, without any synchronization, leading to a wrong result.
Each MPI task spawns two OpenMP threads:

1. Thread $1$ calls the function performing the shared-memory reduce within the intra-node communicator, in which all the tasks are participating.
2. Thread $0$ calls the MPI function to connect master ranks pertaining to different nodes, within the inter-node communicator. Actually only task $0$ calls MPI, for all the others this is a "silent" thread.

With two threads, the master rank coordinates the reduce operation in shared-memory with its siblings in the same node, and manages the MPI communication with all the other nodes' masters. Inter-node communication is not implemented with a ring as well, but still relies on the standard MPI implementation with the non-blocking protocol, i.e. *MPI Ireduce*.
We are developing this implementation further in two ways:

1. We let each MPI task spawn the maximum number of OpenMP threads compatible with the number of CPU cores available. All the threads, with the exception of Thread $0$ (engaged in the communication), are directly involved in the summation.
2. We are implementing a ring-like reduce in inter-node communicator as well by using again MPI shared windows, where each master can access remotely the memory of another node with direct memory access.

The purpose is to speed-up the summation in the intra-node shared-memory reduce, and to find a better reduce implementation to avoid standard MPI calls, which turn out to be bottlenecks when hundreds or thousands computing nodes are used. The hybrid implementation is thought to reduce the communication impact on the runtime, and consequently the energy-to-solution if the code becomes much faster than the pure MPI implementation.

## 3.3 Architectures and energy counters

Intel CPUs are usually constituted by a single NUMA region, meaning that there are not submemory regions pertaining to subgroups of cores, but all over the CPU every core takes the same time to access DRAM. This is particularly useful since Intel CPUs (or sockets) are made by a few number of cores, which in the tests of this chapter will be $12$ and a maximum of $16$ at the Leonardo supercomputer for the results presented in Chapter 4 and Chapter 5. Figure 3.1 is a schematic view of dual-socket Intel computing node. The blue squares represent the two CPUs, which in Intel case each one constitutes a NUMA region. The Intel UPI (Ultra Path Interconnect) connects the two CPUs with a total bandwidth up to $20.4GB/s$ per link. Each socket is connected to the grey rectangles on the edges, which represent the DRAM banks. Again, despite the picture can be misleading, each CPU core takes exactly the same time to access all the memory banks, independently on where it is placed on the socket. PCIe (Peripheral Component Interconnect Express) are at disposal to connect sockets with GPUs, the number of PCIe slots defining the memory transfer maximum bandwidth between the host and the device, which in this specific case is $3 \times 32GB/s = 96GB$, even if one GPU can be connected to only one PCIe. In principle, this node scheme allows to connect $6$ GPUs to the two sockets. Intel Omni-Path Fabric is the connection between the CPU and the network board, which is in turn connected to the other computing nodes. Its bandwith is $100GB/s$.

Figure 3.2 schematically shows how an AMD socket is constituted. Its structure is more complex than the Intel one displayed in Figure 3.1, since here the entire picture would represent one of the blue squares. The socket has $64$ cores in total, divided into four quadrants, which communicate with each other thanks to the I/O die. Each quadrant has its own memory bank, and is connected to it with two MCs (Memory Controllers). Cores pertaining to different quadrants take different times to access

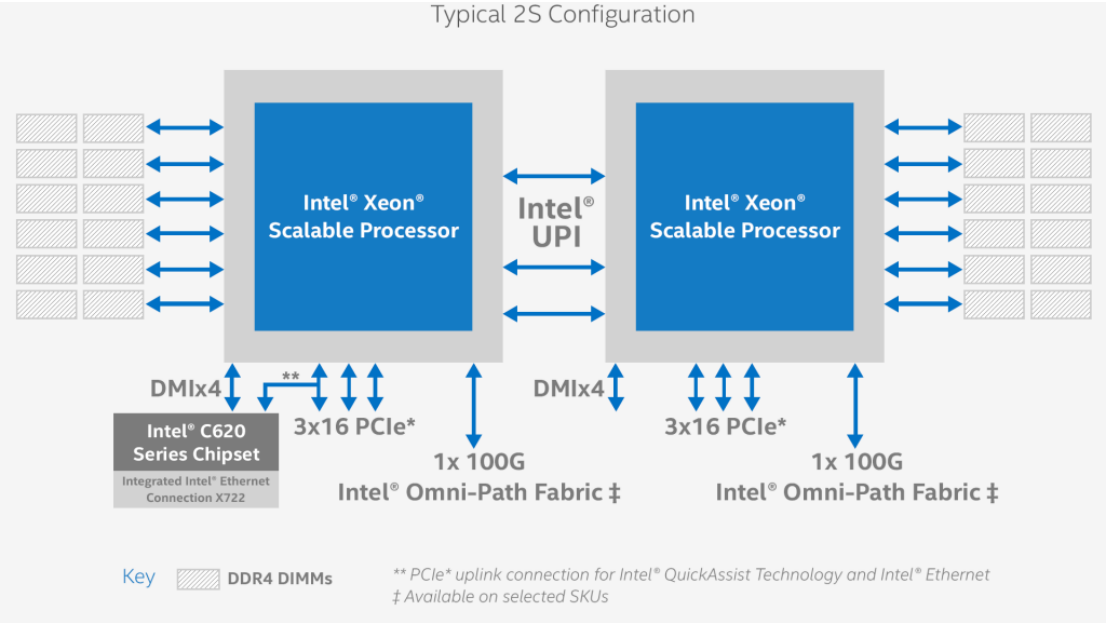



**Figure 3.1:** *Example of a dual-socket Intel architecture. The two heavy blue squares represent the two sockets, each one is a NUMA region. They are connected among each other through the UPI (Ultra Path Interconnect) while at the edges each processor is connected through its DRAM. In GPU nodes, the PCIe (Peripheral Component Interconnect Express) is the main interconnection between CPUs and GPUs, while Omni-Path Fabric is the connection to the inter-nodes network. Intel CPUs can even measure the energy consumption due to memory accesses. Source:* `https://www.chipict.com/intel-purley-platform/`.

the same memory bank. This is the first NUMA level. However, there is a second NUMA level due to the subdivision of each quadrant into four CCD (Core Complex Dies), which in turn contain four CCX (Core CompleXes) with four cores each sharing L3 cache, to get eventually the third NUMA level. This is a really complex NUMA structure, and for really memory intense algorithms, it would be necessary to fill the quadrants in a round-robin fashion, otherwise memory contention will limit the performance significantly, since many cores are forced to access the same (smaller) memory bank in the same time. As an example, by placing Task 0 on first quadrant and Task 1 on the second quadrant, both processes can perform their memory operations independently, since they're actually working on two distinct memory regions. On the other hand, communication involving tasks placed on the same quadrant are faster than ones involving tasks in different quadrants, and the same holds with the finer NUMA regions for CCD and CCX.

In this work I tested the energy efficiency of our implementation, described in Section 3.2, against two different implementations of MPI: OpenMPI and IntelMPI. In particular, we will investigate the performance of the MPI Reduce and the MPI Ireduce, both available in every MPI implementation. The two collectives differ only for the communication protocols, the first one being blocking and the second one being non-blocking.

The tests have been performed on the $2 \times 12$ Intel(R) Xeon(R) Gold 6126 CPU and

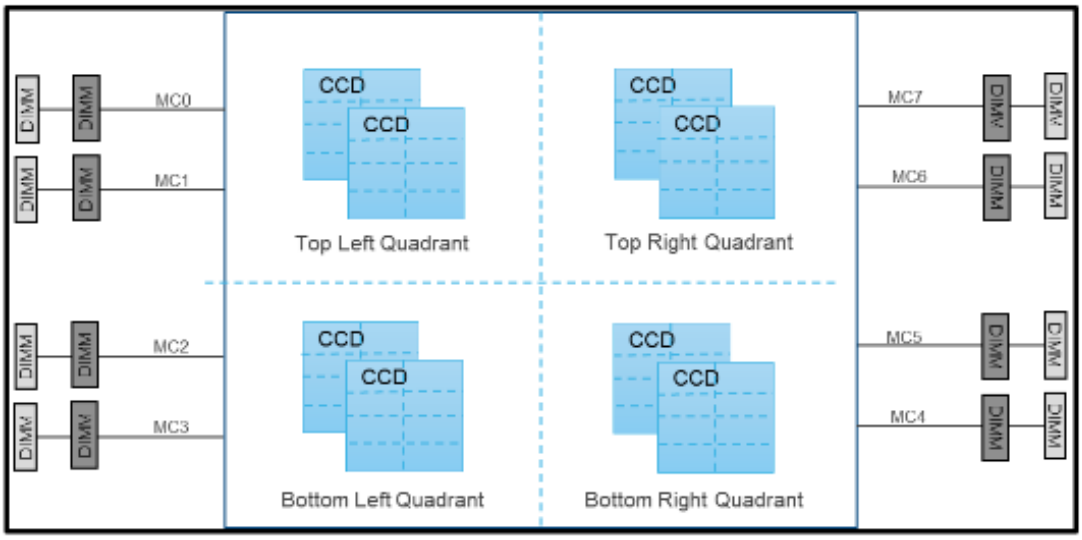



**Figure 3.2:** *Example of a single socket inside a dual-socket AMD Epyc architecture. These sockets have 32/64 cores divided into four quadrants. Quadrants communicate with each other via a central die called I/O die. Inside each quadrant there are two or four CCD (Core Complex Dies) that contain each four CCX (Core CompleXes) each with 4 cores sharing the L3 cache. Each quadrant is connected to its own DRAM through two MC (Memory Controller). To sum up, each CCX is actually a NUMA region, but as we will see later, to maximize the bandwidth it is more useful to assign the MPI tasks to each quadrant in a round-robin fashion, because all the cores inside each quadrant share DRAM. Source:* `https://downloads.dell.com/manuals/common/dell-emc-dfd-numa-amd-epyc-2ndgen.pdf`.

**Table 3.1:** *Technical details of the architectures used in this work. For Intel machines there is one NUMA region per socket (Figure 3.1)., whereas there are 4 NUMA regions per socket exposed by the kernel in this specific AMD architecture (Figure 3.2).*

| Name | NUMA | Cores | RAM |
|---|---|---|---|
| Intel(R) Xeon(R) Gold 6126 CPU | 2 | 24 | 768 GB |
| AMD EPYC 7H12 64-Core Processor | 8 | 128 | 512 GB |

$2 \times 64$ AMD EPYC 7H12 64-Core Processor @ 2.60 GHz computing nodes available at the ORFEO (Open Research Facility for Epigenomics and Other) cluster at the Area Science Park in Trieste, The details of the architectures are listed in Table 3.1.

### 3.3.1 RAPL energy counters

The energy measurements have been performed with the help of RAPL (Running Average Power Limit) counters exposed by the Linux kernel on Intel and AMD architectures (Hähnel et al., 2012; Desrochers et al., 2016; Alt et al., 2024).

I used PAPI (Performance Application Programming Interface) (Mucci et al., 1999; Terpstra et al., 2010; Weaver et al., 2012; McCraw et al., 2014; Wrinkler, 2020) to read the RAPL counters, because it automatically recognizes and how many counters in each socket there are. However, neither the OS nor PAPI actually specify the energy contribution each RAPL counter is actually measuring, it is necessary to be aware of the specific architecture, this means that if both the Intel and AMD counters are exposed by the kernel as one region and one sub-region (per socket) this no way means

that they're measuring the energy consumption of the same event.

For the Intel architecture we have one region measuring the CPU energy of the whole socket, i.e. the energy due to the computation plus other core events like LLC accesses and we have a sub-region measuring the DRAM energy, i.e. the energy consumption due to the memory access. As we will see later, this information is extremely useful because it allows one to study how the memory access impacts in the different implementations (Schöne et al., 2021).

For the AMD architectures we have one region measuring the CPU energy of the whole socket as before, but in this case the energy actually measured is not just due to the core events, but parts of uncore events, like partial DRAM access, are included in the final result. This is a mess because we cannot use the RAPL counters in AMD to estimate the memory access impact as in Intel machines. However, AMD machines permits one to measure per-core energy, i.e. the energy due to the workload of one single core (Schöne et al., 2021).

### 3.3.2 Powercap

PAPI is a tool which allows one to interface with several kinds of hardware counters available on one's CPUs, GPUs, InfiniBand and many others.

These hardware counters can measure cache-misses, instructions-per-cycle, cycle-per-element and also power and energy consumption. To achieve this goal, PAPI has the capability to inspect the powercap counters, thanks to the implementation of **linux-powercap.c**, which accesses to the directory **/sys/class/powercap/** where one can find all the RAPL counters. At the beginning AMD used to monitor the energy consumption through APM (Application Power Management) but then they moved to RAPL counters (Schöne et al., 2021). So both AMD and Intel architectures seem to expose the same RAPL counters, but as already discussed one must be aware that they are actually measuring different energy contributions in different architectures.

### 3.3.3 MSR

Even if RAPL counters are available for both Intel and AMD architectures, which can be exposed by the powercap component, to be more accurate in AMD energy measurements my collaborators and I chose to rely on MSR (Model Specific Register) counters (Design, 1997; Kogler et al., 2022), since they're claimed by the vendor to be more reliable than RAPL.

MSR have access to measure contributions from two different components:

- **CPU energy** This contribution includes the sum of the energies consumed by the single cores, and many other **uncore events**, i.e. cache accesses and partially even DRAM accesses.
- **Core energy** This contribution includes the **core events**, and each core has its own counter. Core energy includes only CPU utilization energy, such that the

summation of all cores contributions is lower than CPU energy, which includes uncore events.

In fact it has been found out that RAPL didn't manage to access all the core counters, such that only the CPU energy could be obtained, losing important information about the single core energy. Furthermore, we noted that even the CPU energy had a bug, especially for multi-socket nodes, where RAPL was able to read the energy counter for one CPU only. With the MSR register this problem is avoided, thanks to a driver able to access Model Specific Registers and creating a folder in which all the energy counters, labelled from 1 to 130, was created. Having the AMD machine at ORFEO 128 cores, labels from 1 to 128 are the core energy counters, while 129 and 130 are the socket 0 and socket 1 total counters, respectively. With this driver we have been able to perform energy measurements on AMD CPUs at both CPU and core levels. Initally the counters weren't able to manage *turn-around*, i.e. they reach a maximum number and then reset, and much effort has been needed to fix this issue.

To make these counters accessible, a pull request with PAPI has been made, such that all users can now perform energy measures with AMD CPUs, and a new branch was created[1]. The PAPI component has then been interfaced with a profiler to read the counters at each step in order to obtain a power profiling.

The profiler is a python code which selects a timestep to call the driver. Apart from AMD MSR, it is able to read RAPL for Intel CPUs and to read the **nvml** (NVIDIA Management Library) counters (Kasichayanula et al., 2012; Raffin et al., 2023; Parashar et al., 2023). However, CPU counters are updated much faster than GPU ones, this setting strong constraints on the timestep choose in the profiler. With CPUs, it was possible to go below $0.05s$ read timesteps to have a finer profiling, while with GPUs going below $0.5s$ led to discontinuous and unreadable results. To use the profiler properly, the user must be aware about the update frequency of the relevant hardware, which may change from CPU to CPU according to vendors.

## 3.4 Results

This section will be devoted to the results presentation for the reduce. Here, only the reduce part will be profiled, instead of the whole radio imaging code, to inspect the efficiency of our new implementation against the standard ones, given by OpenMPI[2] and IntelMPI[3]. As mentioned in Section 3.1, in the first part I'll be focused on the results got in the Intel machine, which allows to profile the energy spent by the CPU and the energy spent to access memory as well. This permitted to study the algorithmic imprint of the reduce operation in CPU utilization and memory utilization, with the focus on energy and runtime differences. It is fundamental to study the relation between energy-to-solution and time-to-solution to understand whether the difference

[1] `https://github.com/NiccoloTosato/papi/tree/amd_energy`
[2] `https://www.open-mpi.org/`
[3] `https://www.intel.com/content/www/us/en/developer/tools/oneapi/mpi-library.html`

in energy consumption is due to the runtime difference only or there are cheaper instructions impacting the energy-to-solution. The energy fraction spent in memory access will be presented and discussed, to grasp the algorithmic memory intensity of the different implementations.

### 3.4.1 Intel

In Figure 3.3 and Figure 3.4 I present the ratios of the homologous quantities, labelled by $X$ in the y-axes, which represents energy or time, between the standard Reduce and Ireduce implementations available in both OpenMPI and IntelMPI and the our ring reduce. Results are averaged over three measurements and standard deviations are chosen as errorbars. Solid lines pertains the OpenMPI library while dashed lines pertains the IntelMPI library.

In the x-axes there are the number of MPI tasks, i.e. what I called $N$. The blue lines are the CPU energies, i.e. the energy needed by the node to perform the computations and other core events like L3 cache accesses (no DRAM access energy is included), the green ones are the DRAM energies, i.e. the energy needed by the node to access DRAM during the execution of the program. It is interesting in this specific case because all the three implementations are memory-bound. The orange lines represent runtimes, i.e. the total times spent by the node to perform the reduce operation. To test the actual reduce efficiencies I have performed one reduce per MPI task and I have kept the size of the problem constant, same number of sums but different memory movings, I performed a strong scalability test by increasing the number of MPI tasks at each step. At fixed problem size, I should expect in the ideal case the allreduce operation to be constant at least in terms of runtime, however the increasing number of reduce operations as a function of $N$ leads to an increase in overhead and consequently to an increase in both energy consumption and runtime. However, notice that the overhead imprint is lower in our implementation compared to the standard libraries, witnessed by the ratio increase as a function of $N$ in both plots.

I notice that our implementation is faster and cheaper in energy than the MPI ones and it's interesting to observe that this ratio does depend on the number of MPI tasks among which the operation is performed. In Figure 3.4 there are some spikes corresponding to $P = 4, 8$, signalling that the MPI Ireduce is implemented in a different way when the number of MPI task is a power of two.

The results of these efficiency ratios are summarized in Table 3.2. Results show that for the Intel architecture ring reduce is significantly faster and cheaper than OpenMPI implementation, up to factors $\sim 6$, $\sim 7$ in CPU utilisation energy and $\sim 4$ in memory utilisation energy. Ring reduce is still faster and cheaper than IntelMPI but with lower gain factors, as can be seen in Table 3.2, meaning that Intel implementation is actually better for these purposes.

Much relevant to notice is that energy ratios and time ratios have a similar shape but they are not exactly superimposed. This means that the energy is not simply an

**Table 3.2:** *Table of ratios between CPU energy/memory energy/runtime of both OpenMPI and IntelMPI and ring reduce for the Intel architecture.*

| Name | CPU | DRAM | Runtime |
|---|---|---|---|
| OpenMPI | $2-7$ | $2-4$ | $2-6$ |
| IntelMPI | $2$ | $1.5-2$ | $2$ |

integral over time, i.e. the energy consumption does not just depend on the runtime, but there are cheaper CPU instructions from one side, and more memory intense algorithms on the other side. This is the main result I want to underline, i.e. different codes doing the same stuff in almost the same amount of time can have different energy consumption due to how CPUs and memory are used.

Figure 3.5 and Figure 3.6 shows the ratios of the blue/green lines and the orange lines of Figure 3.3 and Figure 3.4, in order to infer quantitatively the impact of the algorithmic imprint. Again, solid lines refer to OpenMPI libraries and dashed lines refer to IntelMPI libraries. The blue curves represent the ratios between the CPU energy gain and the runtime gain of our implementation against the MPI standards, and we clearly see that the CPU energy gain is up to $10\%$ more than runtime gain for the MPI Reduce, while it is still above but very close to one for the MPI Ireduce. This means that our implementation uses cheaper CPU instructions than the MPI ones, especially than the MPI Reduce. The opposite happens when the ratios of memory utilisation gains and runtime gains are considered. Our results show that these ratios are well below one signalling that our implementation is more memory intense, as expected. Interestingly, in Figure 3.6 we observe that by summing the two energy contributions I have a number very close to one, meaning that the total energy contribution behaves as an integral over time. To have a confirmation that ring reduce is more memory intense, I plot in Figure 3.7 the ratio between the DRAM energy and the total (CPU + DRAM), which does not diminish trivially as a function of the number of MPI tasks in our implementation as it does happen in a very effective way in the other implementations. Indeed our ring reduce has a memory access energy fraction which stabilizes at about $36-37\%$, while in the MPI Reduce the fraction drops at about $26\%$ and in the MPI Ireduce it drops at $30\%$. This actually is not a great issue because we see in Figure 3.3 that our implementation is up to $3.5$ times cheaper than the MPI Reduce and up to $4$ times cheaper than the MPI Ireduce for memory access, while it's up to $\sim 7$ times cheaper in terms of CPU energy than both the standard implementations. Due to the fact that in all the cases the majority of energy is wasted by the CPU, I can conclude that our ring algorithm in shared-memory is actually more energy efficient than the MPI standards on Intel architectures.

The next steps will be to have a full reduce threadization intra-node so that as many cores as possible can participate in the summation and to extend our ring algorithm on many nodes by implementing a distributed-memory ring among the master ranks of each node.

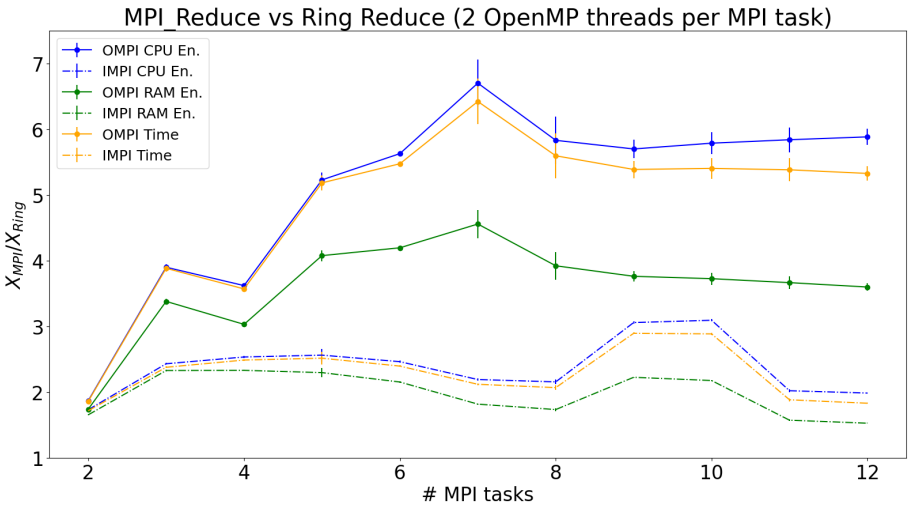



**Figure 3.3:** *Ratio of energy/time of the standard MPI Reduce and our implementation as a function of $N$. Solid lines refer to the OpenMPI implementations, while dashed lines refer to IntelMPI. For this specific architecture we notice that ring reduce is much more efficient both in terms of performance and consumption compared to OpenMPI, whereas the gain becomes more subtle when the direct comparison with the IntelMPI reduce is considered. For each implementation, the curves have almost the same shape, but they are not superimposed. This means that the gain in energy efficiency is not exactly the gain in runtime. Also interesting to notice that the gain I have in memory access energy is not as high as the one in CPU utilisation energy.*

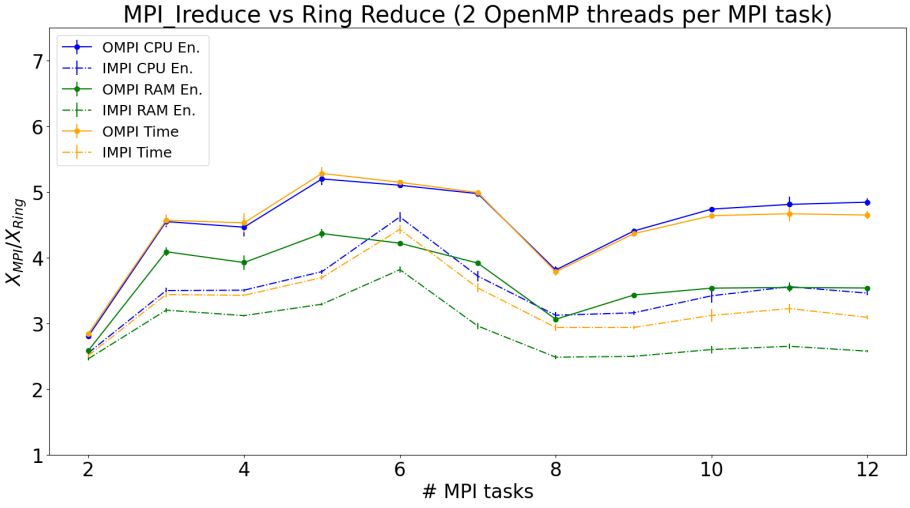



**Figure 3.4:** *Ratio of energy/time of the standard MPI Ireduce and our implementation as a function of $N$. Solid lines refer to the OpenMPI implementations, while dashed lines refer to IntelMPI. For this specific architecture we notice that ring reduce is much more efficient both in terms of performance and consumption compared to OpenMPI, whereas the gain becomes more subtle when the direct comparison with the IntelMPI reduce is considered. For each implementation, the curves again have almost the same shape, but they are not superimposed. As in the MPI Reduce case, this means that the gain in energy efficiency is not exactly the gain in runtime.*

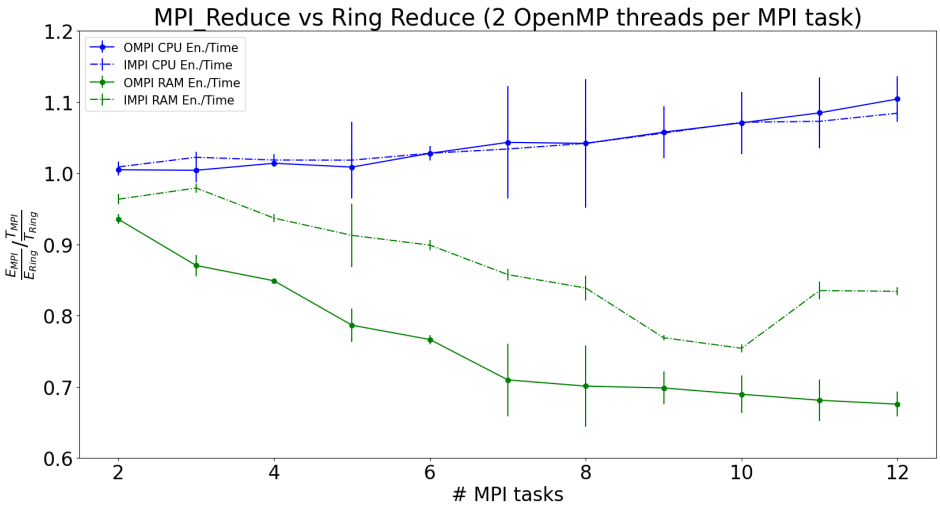



**Figure 3.5:** *Ratios of the CPU/DRAM energy gain and the runtime gain between our ring algorithm and the MPI Reduce as a function of $N$. As expected, for the CPU this ratio is greater than one, signalling that our implementation has cheaper instructions than standards. This is different for DRAM, for which this ratio is less than one, meaning that even the DRAM access energy does not just depend on runtime and the algorithm is more memory intense.*

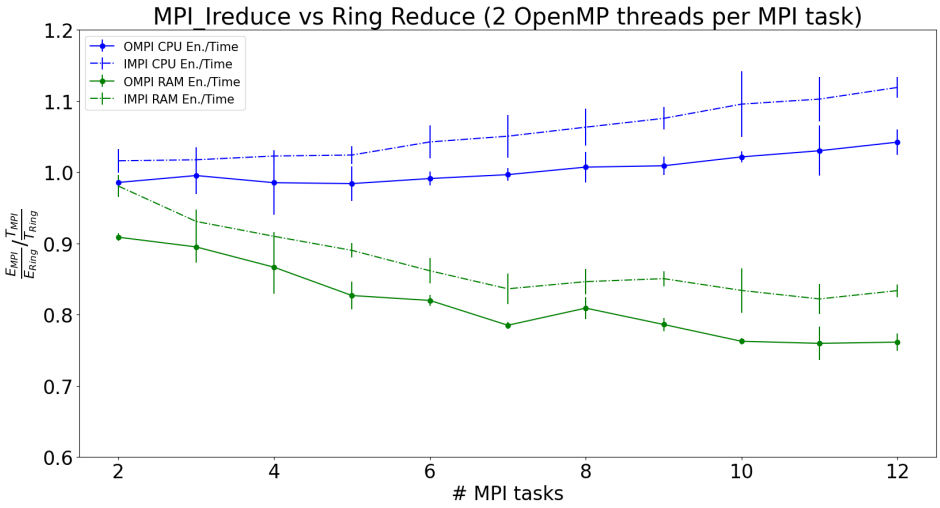



**Figure 3.6:** *Ratios of the CPU/DRAM energy gain and the runtime gain between our ring algorithm and the MPI Ireduce as a function of $N$. The CPU gain now is less than the one from MPI Reduce and the ratio slightly differs from one. As for the MPI Reduce, we still observe that our algorithm is more memory intense compared with the other implementations because again the DRAM/runtime ratio is lower than one.*

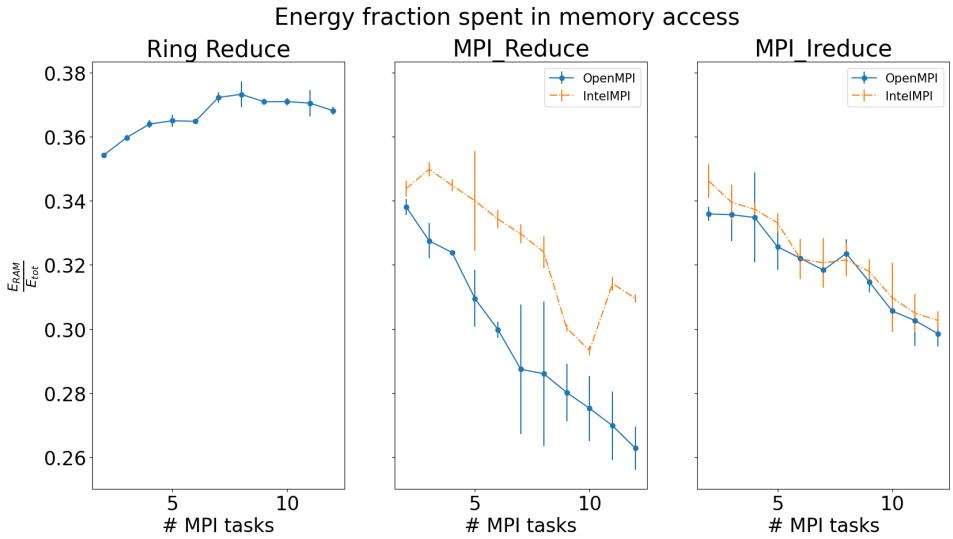



**Figure 3.7:** *Energy due to DRAM access divided by total energy as a function of $N$. Left panel: our implementation has an almost constant energy fraction due to DRAM, but this can also be due to the dimension of L3 cache of the machine we are using. Central panel: MPI Reduce is the implementation which uses the DRAM in the smartest way, because the energy fraction drops up to $\sim 21\%$ when the node is saturated. Right panel: MPI Ireduce uses the DRAM in an intermediate fashion between the other two implementation, but however the energy fraction decreases by augmenting $N$. It will be interesting to inspect how can we use the DRAM in a more efficient way in our ring algorithm.*

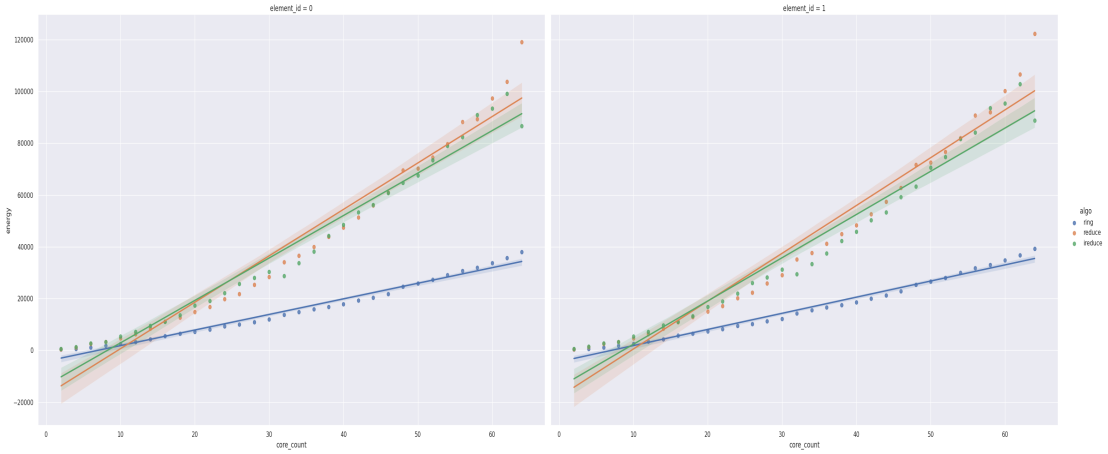



**Figure 3.8:** *Package energy for socket* 0 (*left panel*) *and for socket* 1 (*right panel*) *as a function of* $N$. *Points are numerical data averaged over three runs and solid lines are a linear fit whose shaded areas correspond to standard deviations of the measurements.*

### 3.4.2 AMD

In this paragraph I will show our preliminary results on AMD CPUs, where we did strong scaling tests, again, by keeping the number of summations per task constant and changing the communicated data step by step. In AMD machines we are able to measure CPU energy, labelled also "package", and core energy, where single core counters are read. We expect memory operations to be more expensive than summations, such that the increase in memory operations will be dominant with more and more processes, and the expectation is a linear scaling, since the entire operation is an allreduce at the end of the day.

Figure 3.8 represents the package energy for socket 0 and socket 1, in left and right panel, respectively, as a function of $N$. The energy is now measured in $J$, points are measured data averaged over three runs, while solid lines are linear fit to these data with a shaded area corresponding to the standard deviation of the measurements. From the plot is clear that for few processes per socket the three implementations almost overlap, whereas the main difference is relevant when $N$ grows. The ring reduce energy increases with a smoother slope, which is far more pronounced for the standard MPI implementations. The scatter between green points, pertaining to the Ireduce, and the orange points, pertaining to Reduce, happens when $N$ is a power of two, for which Ireduce turns out to be a more efficient algorithm.

In Figure 3.9 the situation becomes more interesting. Here the energy has been divided by the runtime to obtain an average power for each implementation, as a function of $N$. Indeed ring reduce and Ireduce almost overlap, with the latter doing some "jumps" which reach their minima when $N$ is a power of two. On average, the Ireduce absorbs the same power as the ring reduce, however is much slower, and is impacted by the algorithmic imprint, which turns out to be more efficient for the latter.

Figure 3.10 shows the core energy sum as a function of $N$. This time the ring and

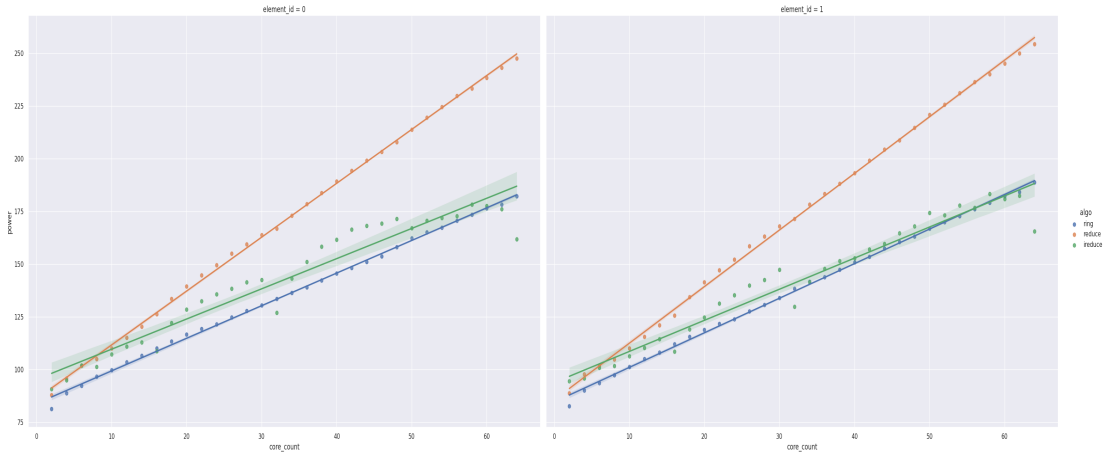



**Figure 3.9:** *Package power for socket* 0 (*left panel*) *and for socket* 1 (*right panel*) *as a function of* $N$. *Points are numerical data averaged over three runs and solid lines are a linear fit whose shaded areas correspond to standard deviations of the measurements.*

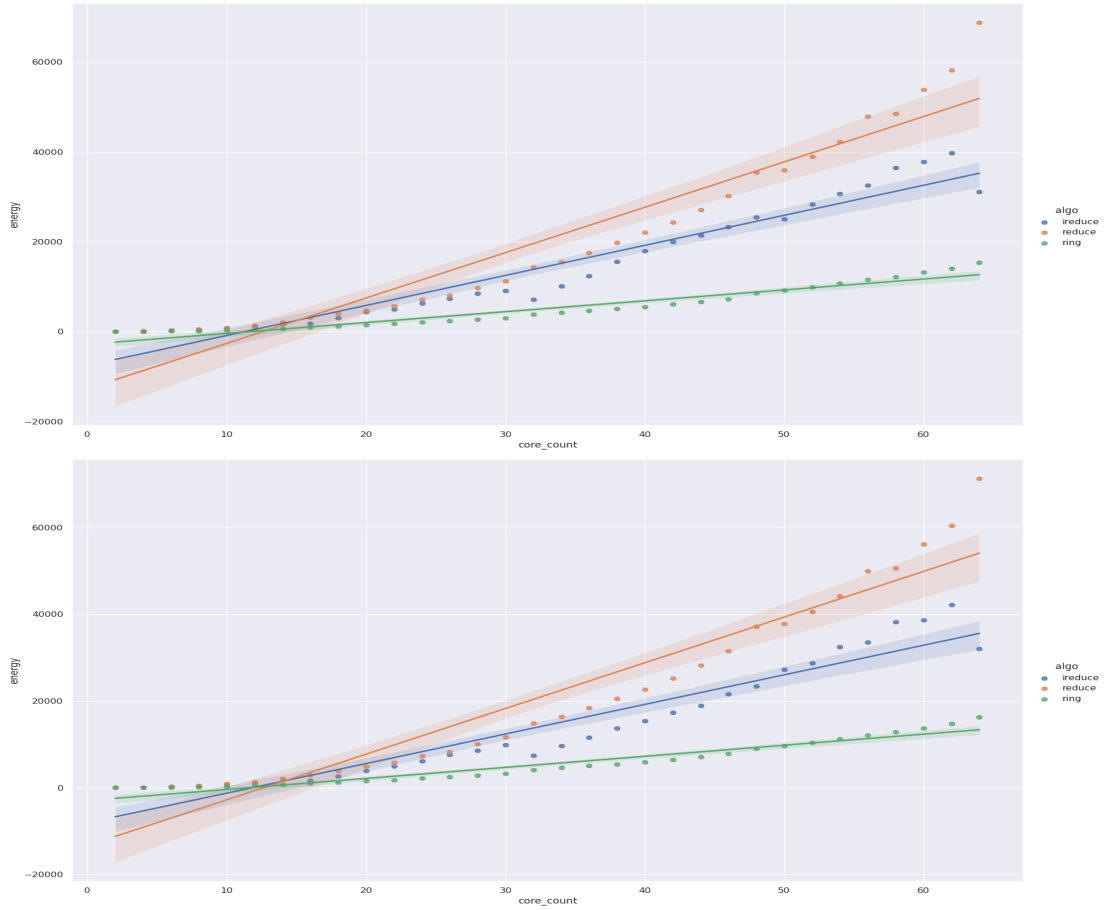



**Figure 3.10:** *Core energy for socket* 0 (*upper panel*) *and for socket* 1 (*lower panel*) *as a function of* $N$. *Points are numerical data averaged over three runs and solid lines are a linear fit whose shaded areas correspond to standard deviations of the measurements.*

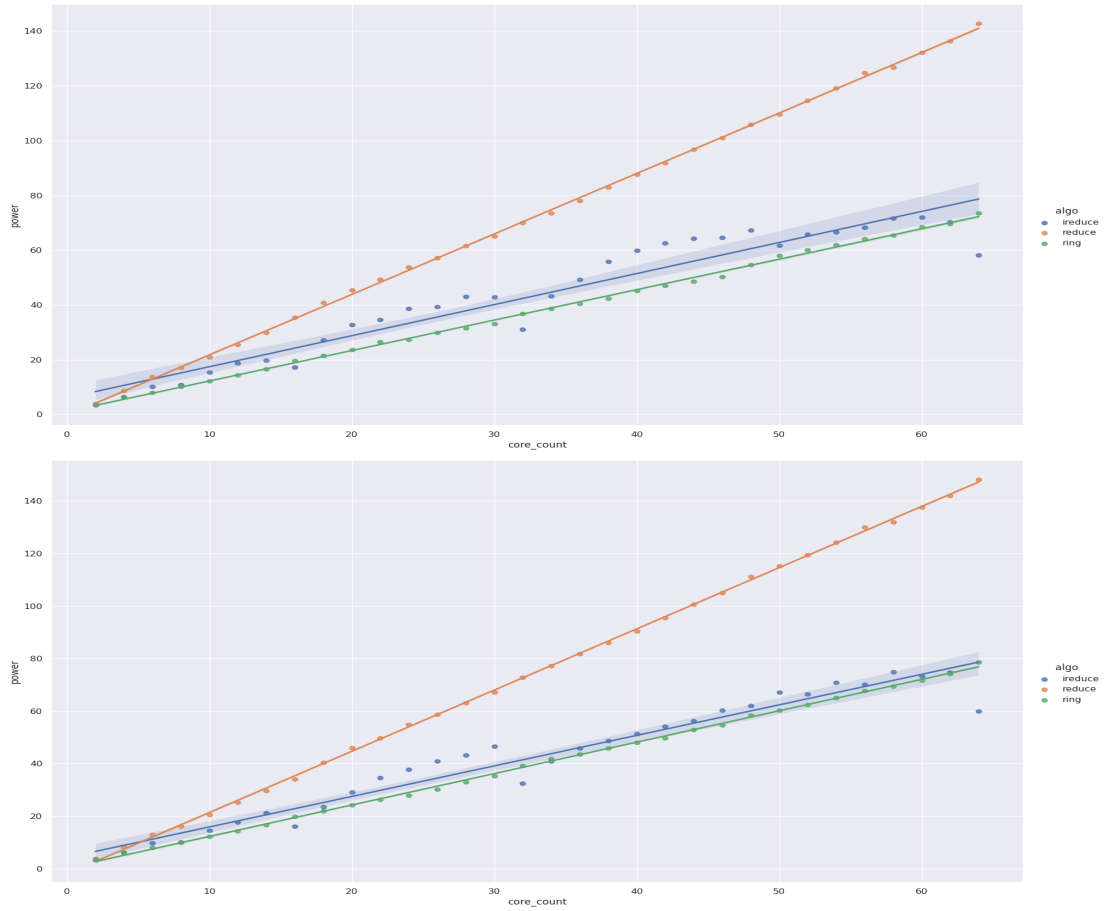



**Figure 3.11:** *Core power for socket* 0 *(upper panel) and for socket* 1 *(lower panel) as a function of* $N$*. Points are numerical data averaged over three runs and solid lines are a linear fit whose shaded areas correspond to standard deviations of the measurements.*

Ireduce colours have been inverted. Again, for small $N$, the three point datasets overlap, with the energy increase becoming less relevant for the ring reduce for larger $N$. However, the difference in energy is less pronounced than in Figure 3.8, until $N \sim 40$. It is again clear that when $N = 32$ and $N = 64$ the Ireduce energy suddenly drops.

The most astonishing result happens in Figure 3.11, when the average power of the three implementations is plotted as a function of $N$. In both panels ring and Ireduce linear fits almost overlap, with a more pronounced effect for socket 1. However, this is true on average, since for non-power-of-two $N$ values the Ireduce data are above the ring ones. However, when $N$ is a power of two, the Ireduce has the minimum absorption.

## 3.5 Conclusions

At first order energy is an integral over time, i.e. if you want to save energy, write faster codes. However, when I plotted the ratio between energy gain and runtime gain in Figure 3.5 and Figure 3.6 I found that there is a non-negligible contribution

from algorithmic imprint. The take-home message is that our ring reduce implementation is not just faster than MPI Reduce/Ireduce, but it turns out to have even cheaper instructions. A low-level code profiling is the next step in the work, in order to inspect which instructions impact the most in energy-to-solution and whether it is possible to port this knowledge in several codes by rewriting them to exploit the most efficient instructions. In terms of memory imprint, our code is more memory intense than MPI, as shown in Figure 3.7. By increasing $N$ the MPI standard implementations start to have a deeper computational impact than the ring reduce. To diminish this memory impact it would be useful to generalize the current hybrid reduce to the fully hybrid reduce, in which all the available threads are used and are involved in the summation. If $N_{th}$ threads are spawned by each process, one will be involved in the communication only and $N_{th} - 1$ will be working in the summation. The generalization to implement a ring reduce among node masters is under development and the results will be presented in a future work. Aside performance, the network ring reduce is necessary since MPI Ireduce is strongly limited by the buffer size. When $N$ is very small and CPUs are filled with OpenMP threads, it happens that the entire buffer must be sent to task $0$, which very often overflows the memory per core available, causing a code crash. The ring reduce will fix this issue again with MPI shared windows, in which data are loaded into the window in bunches, which may be few or even one when data are small, and may be tens or hundreds when data are large. By the way, the current implementation needs to synchronize the window at each bunch, leading to a performance bottleneck which becomes more pronounced when the number of bunches is large.

Preliminary results on the AMD machine reveal that the ring reduce energy dependence is smoother than MPI implementations, however in terms of power absorption Ireduce and ring reduce almost overlap, with the former being even better when $N$ is a power of two. This leads us to change the algorithm for these $N$ values as well, in order to have a logarithmic tree or a reduce-scatter (Iannello, 1997) algorithm.

*This page intentionally left blank.*

# 4

# Single-node and I/O tests

## 4.1 Introduction

The RICK code aims at exploiting multiple CPUs, GPUs and distributed memory parallelism, splitting data and Cartesian 3D computational mesh across different processing units, targeting optimal scalability on large HPC architectures. However, a number of relevant applications in radioastronomy can be accomplished using a single computing node, integrating one multi-core CPU with several GPUs. Specifically, problems targeting images with size below 10000×10000 pixels with input data that can be split by frequency or sliced in time, can be efficiently addressed on individual processing nodes. Running on a single node also allows for seamless integration of the code into conventional data processing pipelines that do not have the capability for parallel processing on multiple nodes.

This study showcases the efficiency of the RICK code when executed on a single computing node. In Section 4.2, we provide a concise overview of the methodology and the experimental setup. In Section 4.3, we discuss the impact of parallel I/O in the code. In Section 4.4 and Section 4.5, we show the results achieved by running the code on various CPU/GPU configurations of the computing node, with small and large configurations. Conclusions are drawn in Section 4.6.

## 4.2 Experimental setup

By utilising a recursive data loading and gridding process, data of any size can be efficiently managed. The tests described in this chapter assume that various frequency bands are stored in separate files, which is a common practice with MeasurementSet files, a common file format in radioastronomy. Once the previous file's data has been gridded, each file is loaded into memory using parallel I/O. Data load parallelism can be also implemented dividing the input data into time-chunks. This feature enables users to adjust the fraction of the processor's memory allocated for input data. The tests presented in this study utilise a dataset obtained from a LOFAR HBA Inner Station 8 hours observation, whose main characteristics are presented in (Gheller et al.,

2023)[1]. The dataset consists of 25 sub-bands of 2 MHz each, spanning a frequency range between 120 and 170 MHz. For our tests, we have used the first 8 sub-bands. Including more sub-bands would only lengthen our tests without contributing any further insights to our discussion. To evaluate the performance of RICK, we have performed a number of tests and benchmarks exploiting the Leonardo HPC architecture, available at the CINECA supercomputing centre. CINECA is the Italian national HPC facility and the Leonardo system is made of 3456 32-cores Intel Xeon Platinum 8358 CPU nodes equipped with 4 NVIDIA Tesla Ampere 100 GPUs. Each CPU node has 512 GB of DDR4 memory, the memory of the GPU is a 64 GB HBM2. The CPU and the GPU are interfaced by a PCIe Gen4 interconnect, while the 4 GPUs per node adopts NVLink 2.0. The systems deploys a Lustre parallel filesystem as working storage area. The source code has been compiled using the GCC and the NVCC compilers and the available MPI library. The GPU tests have been performed using CUDA version 12.1 and NVIDIA HPC SDK version 23.11: this suite includes NCCL version 2.18.5, NVSHMEM version 2.10.1, and cuFFTMp version 11.0.14. Each single input sub-band used for the tests requires about 4.4 GB of memory. Therefore, a significant portion of the CPU/GPU memory can be allocated for the computational mesh. We have executed two series of tests. The first aims at analysing the performance and the scalability of a node using different numbers of cores and GPUs. In order for these tests to be run on a single GPU, the mesh size has been set to $4096 \times 4096 \times 32$ cells. The second aims at stressing the node architecture. We have then adopted the largest mesh that can be managed by the GPUs, give their aggregate memory capacity of 256 GB. This corresponds to $14000 \times 14000 \times 32$ cells.

## 4.3 Data Load

Splitting the data load in frequencies (or times) overcomes the limitations imposed by the CPU/GPU memory size. Data read is accelerated through parallel I/O, each MPI task reading its own portion of input data.

Table 4.1 displays the data load performance in terms of GB/sec, based on the number of MPI tasks. The parallel Lustre filesystem has data caching capabilities, which means that data read multiple times by the same process is kept in memory, allowing for quicker access than from the disc. When a single MPI task is utilised, data is loaded at a rate of 1.4 GB/sec from the disc and 2.3 GB/sec from the cache. As the number of MPI tasks increases, the disc data rate also grows, although not linearly. With four MPI tasks, targeting the four GPUs on the node, the speed-up is of a factor of around 2.5. When cache is utilised, the speed-up approaches linearity due to the prefetch mechanism of the filesystem. Further increasing the number of MPI tasks, we can observe a steady improvement in performance. However, the Lustre filesystem's hardware configuration limits the measured bandwidth from disc to approximately

[1] `https://titulus-inaf.cineca.it`

**Table 4.1:** *Data read performance, measured in GB/sec. The first column represents the number of MPI tasks, while the second and third columns display the performance for direct disc access and Lustre cache access, respectively.*

| MPI tasks | Disk (GB/sec) | Cache (GB/sec) |
|---|---|---|
| 1 | 1.4 | 2.3 |
| 2 | 2.3 | 4.3 |
| 4 | 3.7 | 7.1 |
| 8 | 5.1 | 11.6 |
| 16 | 6.1 | 18.7 |
| 32 | 6.3 | 22.9 |

6.3 GB/sec for 32 MPI tasks, even when all cores of a node are used to read data in parallel. In contrast, the access to cached data results in a substantial improvement in performance, even for 32 tasks.

## 4.4 Small Tests

In a first set of tests we have adopted a computational mesh of size $4096 \times 4096 \times 32$ cells. These configuration fits the memory of a single GPU. In order to evaluate the performance on different multithreading and GPU setups, single core benchmarks are used as the baseline. The results are presented in Table 4.2.

**Table 4.2:** *Time to solution for the Small test in different CPU/GPU setups. Breakdown for the main computational kernels together with the communication overhead are provided. The first column reports the number of MPI tasks. The second column shows the number of CPU threads and the third the number of GPUs used in the test. The following 3 columns present the average times to solution of the main computational kernels. The seventh column reports the MPI overhead, while the last column presents the total time to solution, comprising the contributions of the data input and output and of few additional algorithmic components.*

| MPI Tasks | Threads | GPUs | Gridding [sec] | FFT [sec] | w-stack [sec] | Comm. [sec] | Total [sec] |
|---|---|---|---|---|---|---|---|
| 1 | 1 | 0 | 322.06± 2.95 | 30.01±0.54 | 26.08± 0.03 | 0 | 418.07± 3.25 |
| 1 | 32 | 0 | 12.26± 0.14 | 3.07±0.07 | 2.18± 0.19 | 0 | 48.30± 0.80 |
| 1 | 1 | 1 | 23.29± 0.02 | 0.70±0.01 | 0.1429± 0.0002 | 0 | 68.98± 0.61 |
| 2 | 1 | 2 | 9.51± 0.07 | 0.86±0.10 | 0.0781± 0.0030 | 1.80±0.01 | 47.34± 1.05 |
| 4 | 1 | 4 | 4.53± 0.05 | 0.53±0.01 | 0.0403± 0.0008 | 1.51±0.22 | 23.67± 0.44 |

It is evident that the utilisation of multithreading and GPU acceleration is crucial in significantly decreasing the time to solution. The multithreading implementation achieves a significant speed-up of 8.7 times compared to a serial run on a single core. This improvement is achieved by utilising all 32 available cores. Furthermore, when the GPUs are utilised, the speed-up is even higher, reaching almost 20 times. By utilising all the GPUs, the performance is approximately doubled compared to the 32

threads CPU. This limited improvement in performance can be attributed to the inclusion of data load, which does not benefit from GPU acceleration, as well as some algorithmic components that have not yet been accelerated on the GPU.

When examining the timing breakdown of the various kernels, it becomes clear that the GPU acceleration is remarkable for the FFT and w-stack kernels. The speed-up achieved is more than 4 times for the FFT kernel and over 14 times for the w-stack kernel, when comparing the performance of a single GPU to that of the 32 threads CPU. The speed-up is larger when 2 and 4 GPUs are used. The GPU acceleration of the gridding kernel appears to be much less effective, with one GPU taking nearly twice as long as the 32 CPU threads to perform the gridding. This is expected to be due both to the low arithmetic intensity of the gridding kernel and to the frequent race conditions when updating cells values by multiple threads, both affecting the GPU implementation. MPI communication, present only for GPU setups, is managed within a single node through the hi-speed NVLink interconnect, hence it does not introduce a relevant overhead to the computing time.

## 4.5 Large Tests

Large tests uses the same input dataset as above, but with a computational mesh of size $14000 \times 14000 \times 32$ cells. This is the largest configuration we can run on the node distributing the input data and the computational mesh evenly across the four GPUs.

**Table 4.3:** *Time to solution for the Large test using the full node capabilities in terms of CPU, with multithreading, and GPU, using the four available accelerators. Breakdown for the main computational kernels together with the communication overhead are provided. The first column reports the number of MPI tasks. The second column reports the number of CPU threads and the third the number of GPUs used in the test. The following 3 columns present the average times to solution of the main computational kernels. The seventh column reports the MPI overhead, while the last column presents the total time to solution, comprising the contributions of the data input and output and of few additional algorithmic components.*

| MPI Tasks | Threads | GPUs | Gridding [sec] | FFT [sec] | w-stack [sec] | Comm. [sec] | Total [sec] |
|---|---|---|---|---|---|---|---|
| 1 | 32 | 0 | 12.17± 0.02 | 41.43±0.10 | 19.44± 0.69 | 0 | 107.31± 5.87 |
| 4 | 1 | 4 | 5.38± 0.01 | 1.22±0.01 | 0.47± 0.01 | 4.53±0.04 | 30.11± 0.41 |

The GPU exhibits a significant increase in speed, approximately 3.5 times faster than the CPU's multithreaded performance. The performance difference between the larger tests and the smaller ones can be attributed to the reduced influence of data load and non-accelerated parts of the code on the overall time to solution. In addition, the GPUs demonstrate superior performance in both the FFT and w-stack kernels when compared to the CPU. Specifically, in these tests, the GPUs outperform the CPU by factors of more than 30 and 40, respectively. For the gridding part, the same considerations of Section 3.2 hold. Gridding performance is even further penalised due to the larger size of the computational mesh resulting in more frequent main memory

accesses.

## 4.6 Conclusions

This chapter explores the performance of the RICK code in radio astronomy imaging using a high-performance computing (HPC) approach. Specifically, we focus on parallelism and acceleration through GPUs when utilising a single hybrid computing node equipped with a 32-core CPU and 4 GPUs. The code has the ability to split input data either by frequency or time and also supports parallel I/O.

The main outcomes can be summarised as follows:

- The exploitation of all the capabilities of computing nodes provides outstanding acceleration to the imaging process. Compared to a serial code, RICK runs more than 8 times faster using all the available cores and more than 18 times faster using the GPUs.
- Working with datasets of any size is made possible through the use of data input split. Efficient parallel I/O is crucial for accelerating the data load process and overcoming the limitations posed by large data sizes and disc access.
- The MPI implementation is essential for the purpose of enabling parallel I/O, running on multiple GPUs, and, most importantly, dividing datasets among various memories to facilitate the processing of large use cases that would otherwise be unfeasible.
- The FFT and the w-stacking algorithms are extremely efficient on the GPU. The gridding algorithm suffers by low arithmetic complexity and frequent race conditions. It is essential to have all the code components on the GPU in order to ensure the maximum efficiency, avoiding overheads due to data movements from and to the GPU.

The RICK code is currently in the process of being developed to enable the optimal exploitation of large HPC configurations, which can consist of hundreds of GPUs or thousands of cores. This will be the subject of the next chapters.

*This page intentionally left blank.*

# 5

# Large scalability tests

## 5.1 Introduction

We present the results obtained on a state-of-the-art supercomputing platform, namely the Leonardo pre-exascale system operated by CINECA, the Italian National Supercomputing Centre, ranked as seventh in the TOP500 list[1] of June 2024. CPU and GPU tests have been performed using the same code base switching between different building options selecting the proper flags and macros in the Makefile and exploiting the NVIDIA SDK available on the system. All tests have been performed using LOFAR datasets, representative of current radio-observations. The chapter is organised as follows. In Section 5.2 we introduce the solutions adopted for the full GPU enabling of RICK. In Section 5.3 the results of the performance and scalability tests are presented and discussed. Section 5.4 is devoted to the communication and to its impact on the code. Conclusions are drawn in Section 5.5.

## 5.2 Multi/Many Core Based HPC Architectures Support

In this Section, we discuss the newly implemented features of RICK, including the support for multi-many threads FFT, the integration of MPI based direct GPU-GPU communication and the introduction of an optimised hybrid shared-distributed memory to perform the reduce operation in substitution of the standard MPI library calls.

### 5.2.1 Parallel FFT on the GPU

Running the FFT directly on the GPUs represents a pivotal step for the RICK code since it reduces the computational cost of this operation and removes the data movement associated with it (see (Gheller et al., 2023)).

We have adopted the NVIDIA CUDA FFT library (cuFFT[2]), which provides an interface for computing FFTs on an NVIDIA GPU. In particular, given our distributed

[1] `https://www.top500.org`
[2] `https://docs.nvidia.com/cuda/cufft/contents.html`

memory parallelism approach, we exploited the distributed version of the cuFFT, namely the cuFFT Multi-process library (cuFFTMp[3]). This is a multi-node, multi-process extension to cuFFT that supports multiple GPUs across multiple nodes, a key feature given that large size problems hardly fit into a single GPU memory. At the time of writing, cuFFTMp is the only library capable of solving the FFT problem among distributed GPUs. Specifically, we have used version 11.0.14 of the cuFFTMp, included in version 23.11 of the NVIDIA HPC SDK toolkit. The cuFFTMp library uses NVSHMEM[4], a communication library based on the OpenSHMEM standard that creates a global address space that includes the memory of all GPUs in the cluster. The main steps of the cuFFTMp implementation of the 2D FFT on NVIDIA GPUs are listed hereafter.

- A plan for 2D complex-to-complex FFT is created, which contains all information necessary to compute the transform, including the pointers to the input and output arrays.
- For each $w$ plan, distributed data are copied to the respective *descriptor*, that represents an ad-hoc structure for data that have to be transformed.
- The previously-initialised plan is executed doing a 2D inverse Fourier transform.
- Data are redistributed to the original order with a DEVICE_TO_DEVICE copy, because after the FFT they are distributed in a permuted order.
- Finally, transformed data are brought back from descriptors and written into a new device pointer, always distributed among multiple GPUs, which is then used for the following steps of the code.

This operation is executed for each $w$ plan, as required by the $w$-stacking algorithm. Within the loop over $w$, we need to allocate and free device memory for the first descriptor used by the cuFFTMp, that is the one used for the execution of the FFT plan. A second descriptor is instead used only to redistribute data in the original order, so it can be allocated and freed just once outside the $w$-layers loop. The number of layers to be transformed has a significant impact on the overall performance of this step. Also the initialisation of the library, in particular with the creation of the plan at the beginning of the FFT step, impacts the computing time. This will be further discussed in Section 5.3.2.

Having data resident on the GPU memory, we managed to use a CUDA kernel to write the to-be-transformed input array into a *cufftDoubleComplex* data type, dividing the real and imaginary parts, and copying it directly within the descriptor, speeding up its creation and optimising the requirement of adopting this kind of data structure. We also used a CUDA kernel to write FFT transformed data back to the right pointer on the GPU which is then used for the following the $w$-correction step. This fully eliminates the communication between host and device, hence preventing the need for data transfer between the CPUs for every process and $w$-plan.

[3] `https://docs.nvidia.com/hpc-sdk/cufftmp/index.html`
[4] `https://developer.nvidia.com/nvshmem`

### 5.2.2 Hybrid CPU-based FFT

In the code presented in (Gheller et al., 2023), the FFT algorithm could exploit only the MPI implementation (FFTW-MPI[5]). When GPUs are used, each of them is assigned to a single MPI task, running on a single CPU core. These cores are the only ones that can contribute to a pure MPI FFTW calculation: therefore, a significant fraction of the node's computational capacity is wasted. In order to overcome this limitation, we have exploited the MPI+OpenMP parallelism, using the Hybrid FFTW[6] which allows to combine MPI tasks and OpenMP threads to fully utilise the CPU as follows.

- First, data are distributed among all the MPI tasks as in the FFTW-MPI implementation.
- Then, each MPI task spawns a certain number of OpenMP threads for a further workload distribution.

This allows exploiting all the available hardware: in the previous example, the hybrid FFTW runs on 4 MPI tasks, each task spawning 8 threads, effectively maximising the CPU performance. In addition, when all the cores of each CPU are used, the hybrid implementation is expected to scale better compared to the pure MPI one, since the number of MPI tasks performing communication is reduced, leading to a lower communication surface. Communication is further optimised adopting the strategies discussed in Section 5.2.3.

### 5.2.3 Communication

The reduce operation consists in summing contributions gathered from all the MPI tasks in the memory of a target processor or GPU. This can introduce a significant overhead even on a single node when the number of MPI tasks per node is large or the problem size increases (see (Gheller et al., 2023) for details).

To reduce such overhead, we have adopted two different solutions. First, we support direct GPU-GPU communication, which allows $i$) exploiting the NVlink NVIDIA high-speed interconnect within a node, $ii$) avoiding the CPU-GPU data transfer necessary for standard MPI communication. Second, we exploit MPI and OpenMP to combine shared and distributed memory data access in order to $i$) optimise the access to local data, $ii$) minimise the number of MPI tasks and related message passing communication overheads.

**GPU-based Reduce.** Direct GPU-GPU communication has been implemented by exploiting the NVLink high-speed interconnect for NVIDIA and InfinityFabric, the corresponding technology available for AMD architectures. NVLink is a wire-based communications protocol that can be used for data and control code transfers in processor systems between CPUs and GPUs and solely between GPUs. The Leonardo Booster

[5] `https://fftw.org/doc/Distributed_002dmemory-FFTW-with-MPI.html`
[6] `https://www.fftw.org/fftw3_doc/Combining-MPI-and-Threads.html`

is equipped with NVLink 3.0, which provides a transfer rate of 50 Gbit/s and a bandwidth per GPU of 600 GB/s.

We refer to NVIDIA Collective Communication Library (NCCL[7]) which implements the reduce operation as a ring intra-node, and an inter-node ring, when GPUs assigned to the main tasks communicate with Remote Direct Memory Access (RDMA) with GPUs in different nodes without passing through the CPUs (A. Li et al., 2019). A similar solution can be adopted for AMD GPUs thanks to ROCm Collective Communication Library (RCCL[8]), by simply replacing CUDA calls with HIP calls.

**Hybrid CPU-based Reduce.** To mitigate the impact of reduce operations, we have designed a hybrid reduce technique which combines MPI and OpenMP parallelism and exploits the Non-Uniform Memory Access (NUMA) topology. This hybrid implementation works as follows: as first, two MPI communicators are built: 1) an intra-node communicator, where only the MPI ranks pertaining to each computing node are included and are ranked from 0 to $P-1$, with $P$ being the number of MPI ranks on every node, 2) an inter-node communicator that groups all ranks 0 in the intra-node communicators. In communicator 1, each MPI task knows its siblings and exchanges data using shared-memory windows with a ring algorithm. Each MPI task spawns two threads. Thread 1 is in charge of calling the shared-memory reduce function with a ring fashion involving all the other tasks in the intra-node communicator with its siblings, while thread 0 manages the MPI communications among the nodes. It is important to say that although each task needs at least two threads, only task 0 (i.e. the master) is actually involved in the MPI communications. So in the current implementation all the other tasks spawn "silent" threads. In a future implementation we plan to fully occupy the CPU cores with OpenMP and let all the threads participate to the summation. When the shared-memory reduce is done, task 0 gathers the result: this is the total summation in case of a single node run, and it is a partial summation in multi-node cases. The partial sums on each node are finally added using an MPI Ireduce call among the tasks in communicator 2, resulting in the overall sum. Hereafter, when we discuss results about hybrid MPI+OpenMP tests, we assume that the hybrid reduce is used.

## 5.3 Performance Analysis

To evaluate the performance of RICK, we run a number of tests exploiting the Leonardo HPC architecture, available at the CINECA supercomputing centre. The system interconnect is a Nvidia Mellanox network, with Dragon Fly+, capable of a maximum bandwidth of 200Gbit/s between each pair of nodes. The systems deploys a Lustre parallel file system as working storage area. The source code has been compiled using the GCC and the NVCC compilers and OpenMPI library v4.1.4. The only dependency of the code is on the FFTW3 library. The Makefile requires only few adjustments of

[7] `https://developer.nvidia.com/nccl`
[8] `https://rocm.docs.amd.com/projects/rccl/en/latest/`

| | LOFAR Dutch | LOFAR ILT |
|---|---|---|
| RA | 15:58:18.96 | 17:12:50.04 |
| Dec | +27.29.19.20 | +64.03.10.60 |
| Observation time | 8 hrs | 8 hrs |
| Integration time | 4sec | 4 sec |
| N. antennas | 62 | 72 |
| Bandwidth | 1953.1 kHz | 1953.1 kHz |
| Reference frequency | 150.5 MHz | 144.6 MHz |
| N. sub-band | 1 | 1 |
| N. of channels per sub-band | 20 | 160 |
| Channel width | 97.656 kHz | 12.207 kHz |
| Project code | LC14_018 | LT16_005 |
| Principal Investigator | F. Vazza | A. Botteon |

**Table 5.1:** *Main characteristics of the LOFAR HBA datasets used for the Small (LOFAR Dutch) and Intermediate/Large (LOFAR-ILT) configuration test.*

several variables (e.g. the path to FFTW or to the CUDA libraries, the gridding kernel, the enabling of GPUs) to compile the code. The building procedure requires only a few seconds. The GPU tests have been performed using CUDA version 12.1 and NVIDIA HPC SDK version 23.11: this suite includes NCCL version 2.18.5, NVSHMEM version 2.10.1, and cuFFTMp version 11.0.14.

Three different input data configurations have been used for the tests, namely the *Small*, *Intermediate* and *Large*, addressing different test sizes. The first can run on a single CPU/GPU up to a small number of computing units. This allows for testing the performance of each unit and comparing the scaling within a node and between nodes. The second encompasses a number of intermediate computational configurations, that can be frequently expected for radio astronomical data. The third focuses on extreme cases, for which large images have to be generated and hundreds of computing units are required to ensure that enough memory is available to manage the computational mesh. The input data comes from two different radio measurement sets (MS) from LOFAR 8 hours, dual inner mode, observations in high band antenna (HBA). One of the MSs is only constituted by the Dutch stations (LOFAR Dutch), the other using the full International LOFAR Telescope (LOFAR ILT). The main characteristics of the two observations, together with the size of the computational mesh adopted in the three test cases, are presented in Table 5.1 and 5.2. The dataset for the *Small* tests is a single sub-band of 2 MHz of the LOFAR Dutch observation at 150 MHz. The *Intermediate* and *Large* tests used instead the same LOFAR ILT MS, with the only difference residing in the size of the computational mesh.

For each test, we report the following wall-clock time measurements related to specific code components:

- the gridding time, which is the time required for the gridding operation on the visibilities;
- the reduce time, related to the reduce operation;

| | Small | Intermediate | Large |
|---|---|---|---|
| N. visibilities (approx) | $0.54\times10^9$ | $47.05\times10^9$ | $47.05\times10^9$ |
| Input data size (GB) | 4.4 | 533 | 533 |
| $N_u = N_v \times N_w$ | $4096^2\times16$ | $16384^2 \times 32$ | $65536^2\times32$ |
| Mesh size (GB) | 10.81 | 258.81 | 4098.81 |

**Table 5.2:** *Configuration and computational mesh used in the Small, Intermediate and Large tests. The mesh size takes into account the total amount of memory required for the real and imaginary part depending on the size of the grid.*

- the FFT time, which is the time required for inverse Fourier transform of the gridded visibilities;
- the $w$-correction time, the last step of the code before writing the final outputs;
- the total parallel time, which is the total execution time of the code neglecting the initial setup time, which involves mainly I/O data load operations and the writing of the final image.

In order to assess the performance of the different code components, we have measured the wall-clock times they took to complete the different tests and we have discussed the best performance for each configuration. We have then analysed the code scalability, that can be defined as the ability to handle more work as the size of the computing system or of the application grows. Scalability has been assessed in terms of strong and weak scaling. The former measures the performance of the code keeping the data and the mesh size constant and progressively increasing the adopted computing units. Ideally, the code execution time should scale linearly with the number of computing units (i.e. the time should halve if the number of computing units doubles). However, various factors can impact such an ideal behaviour, leading to a performance degradation with increasing the number of units. Therefore, strong scalability indicates the gain the user can expect increasing the number of computing units working cooperatively.

The strong scalability can be measured by the *speed-up* parameter, calculated as

$$S = \frac{T_1}{T_N} \tag{5.1}$$

where $T_N$ is the wall clock time measured using $N$ computing units and $T_1$ is the corresponding time measured using a baseline configuration (e.g. a single computing unit).

In the case of weak scalability, both the number of processing units and the problem size are progressively increased, resulting in a constant memory and work load per processing unit.

### 5.3.1 Code performance

In this section, we provide the results for each configuration to achieve the following objectives: *i*) inform the user about the expected runtime of the code in various setups;

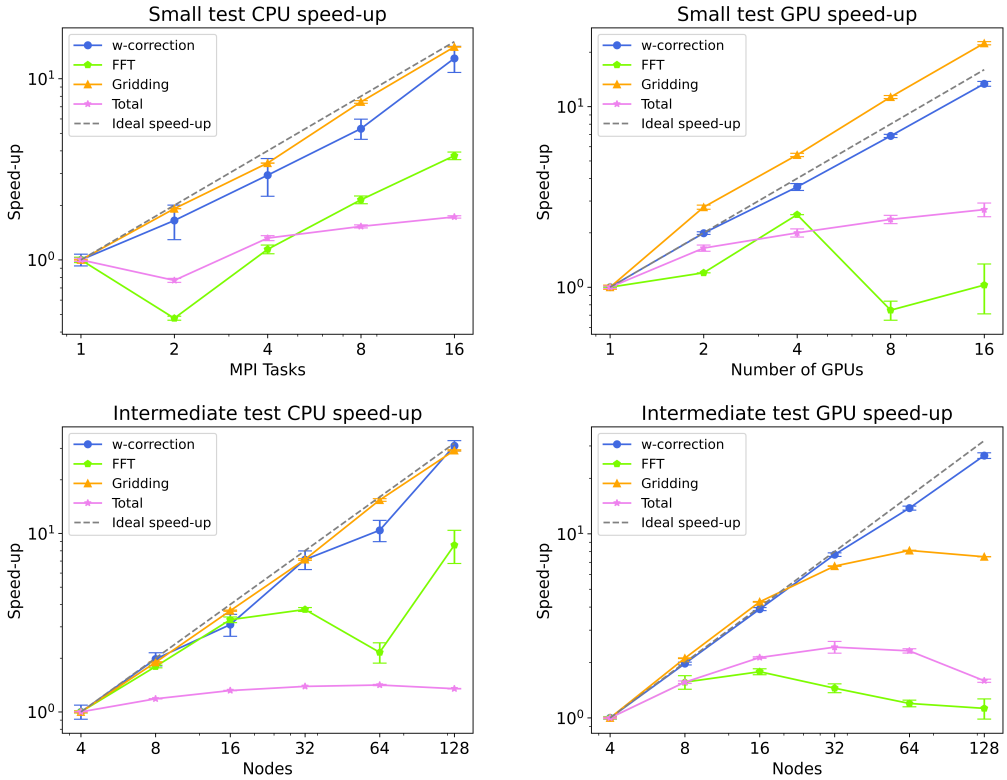



**Figure 5.1:** *Speed-up results for each step of the code for both full CPU and full GPU code, using Small (top) and Intermediate (bottom) dataset. The dashed grey line represents the ideal scaling trend. For the Intermediate test, the number of resources per node is of 4 MPI tasks and 8 OpenMP threads for the full CPU and 4 GPUs and 8 OpenMP threads for the full GPU test.*

| | Nodes | MPI(threads) | GPUs | Gridding (s) | Reduce (s) | FFT (s) | $w$-correction (s) | Total (s) |
|---|---|---|---|---|---|---|---|---|
| *Small* CPU | 4 | 16 (8) | 0 | 0.304±0.001 | 4.351±0.002 | 0.676±0.028 | 0.179±0.028 | 5.636±0.039 |
| *Small* GPU | 4 | 16 (8) | 16 | 0.104±0.001 | 0.207±0.001 | 0.615±0.04 | 0.010±0.001 | 1.187±0.048 |
| *Intermediate* CPU | 64 | 256 (8) | 0 | 1.719±0.026 | 138.352±0.064 | 9.646±1.253 | 0.419±0.050 | 150.944±1.256 |
| *Intermediate* GPU | 32 | 128 (8) | 0 | 1.244±0.026 | 10.132±0.436 | 1.803±0.058 | 0.021±0.001 | 15.965±0.440 |

**Table 5.3:** *Best performance times for Small and Intermediate tests both full CPU and full GPU.*

*ii*) compare the CPU and GPU time to solution. For each configuration, we showcase the best possible result achieved by varying the number of processing units. The timings encompass all the various stages involved in processing the data, including the time spent for communication and for a minor algorithmic component that has not yet been parallelised. Data reading and writing is instead not addressed here. Additional information on the single CPU/GPU performance can be found in (Gheller et al., 2023).

Best performance is presented in Table 5.3 for the *Small* and *Intermediate* tests, while for the *Large* test it corresponds to the set-up with 64 and 32 nodes for the CPU and GPU cases respectively, reported in Table 5.4.

For the *Small* configuration the best result is obtained using 4 nodes for both the CPU and the GPU tests. Both tests run with 4 MPI tasks per node, the former spawning 8 OpenMP threads per task, and latter assigning one GPU per MPI task. The total

time to solution is around 5.6 seconds for the CPU setup. The GPU setup runs around 4.5 times faster, in about 1.2 seconds. The breakdown in the different algorithmic components shows that gridding and $w$-correction are around 3 and 18 times faster on the GPU respectively. The FFT times are instead comparable. The big difference emerge for the MPI reduce time which is more than 20 times faster for the GPU setup. We will discuss in details the impact of the reduce in the following sections.

The *Intermediate* configuration show the best performance on 64 and on 32 nodes for the CPU and GPU setups respectively. Overall, the GPU setup takes around 16 seconds to complete, almost 10 times faster than the CPU one. In both setups the majority of the time is spent in the reduce operation, while the FFT becomes much faster on the GPU, thanks to the the parallel cuFFTMp implementation, much more efficient than the hybrid FFTW.

For the *Large* tests, the GPU setup runs on 32 nodes around 150 times faster than the CPU one, achieved on 64 nodes. Such result is mainly due to the huge difference in the reduce time. Once more, a dedicated discussion on this aspect is provided below. The breakdown shows that gridding times on GPU and CPU setups are similar, with CPU time faster around 1.4 times than GPU one. We will discuss the reason of this behaviour in the next section. The breakdown shows that FFT times on GPU setup are much faster than in *Intermediate* tests, around 25 times compared to CPU setup.

### 5.3.2 Strong scaling tests

Strong scaling results are shown in Figure 5.1, where the speed-up is measured as a function of the number of computing units for the *Small* and *Intermediate* configurations. In Table 5.4 we report the execution times for the *Large* test. The tests have been conducted on a progressively larger number of computing units, varying in a range that depends on the test size. The minimum number of computing units is set so that the problem can be fit within the available memory. The maximum number of computing units is chosen to guarantee that the computing time is higher than the communication time, with the exception of the *Large* test, which will be discussed separately

The *Small* tests have been performed using up to 4 nodes with 4 MPI tasks/node for the MPI runs (up to 16 MPI tasks in total) or 4 GPUs/node for the GPU runs (with 1 MPI task associated to each GPU), with 8 OpenMP threads. *Intermediate* tests range between 4 and 128 nodes, with a fixed number of 8 OpenMP threads and 4 MPI tasks (or GPUs) per node. The *Large* tests required at least 32 nodes because of the large memory request. Computing resources increased progressively up to 128 nodes, which means a total of 4096 MPI tasks for pure CPU runs or 512 GPUs for the CUDA runs.

**Gridding time.** For the GPU version of the gridding algorithm, the timings account also for the offload time of visibilities and relative weights from CPU to GPU through asynchronous memory copies. The gridding kernel is called iteratively, once per each mesh sector (see (Gheller et al., 2023) for details), so we report here the sum of the

times spent for each individual sector (namely the total gridding time).

For the *Small* tests, Figure 5.1 top panels, we first perform a baseline test using 1 MPI task and 8 OpenMP threads. All the other CPU runs double each time the number of MPI tasks, keeping 8 OpenMP threads per task. We notice the almost perfectly linear scaling of the gridding over a single node, ranging from 1 to 4 MPI tasks. Since each node has 4 GPUs, we have not explored other hybrid single node configurations any further. The tests with 8 and 16 MPI tasks have been performed using 2 and 4 computing nodes, respectively, with 4 MPI tasks per node and 8 threads. The GPU tests for the *Small* dataset show a good scaling starting from 1 GPU up to 16 GPUs distributed again into 4 Leonardo booster nodes. GPUs result to be $\simeq 2-3$ times faster than CPUs (see Table 5.3).

For the *Intermediate* strong scaling tests (Figure 5.1, lower panels), we measure the speed-up of the hybrid MPI+OpenMP implementation for CPUs (bottom left panel), observing that gridding scales almost ideally even up to 128 nodes. In this case a thread synchronisation is needed because different threads of each MPI task have to perform gridding in concurrent regions of the mesh. Anyway, in Intel CPUs, this multi-threading synchronisation, which can lead to an overhead, does not impact on the code performance and the algorithm scales linearly. For the GPU case (bottom right panel), we observe that the gridding shows an ideal scaling only up to 16 nodes, meaning that from 32 to 128 nodes the scaling is not linear anymore. This behaviour appears also for the *Large* tests and it will be discussed further at the end of this Section.

For the *Large* case (see Table 5.4), the CPU tests are pure MPI, running on 32, 64, 128 nodes with 1024, 2048, 4096 MPI tasks, respectively (meaning 32 MPI tasks per node). Gridding time scales linearly when we double the computing resources, due to the fact that no communication overhead is included. For the GPU tests the gridding time does not scale increasing the number of accelerators from 128 to 512, remaining approximately constant. This trend is similar to what was found in the *Intermediate* tests when more than 16 nodes were used. This behaviour is due to the overhead related to CUDA GPU memory management, and, more specifically, to repeated*cudaMalloc* and *cudaFree* calls that are implied by the iteration procedure through mesh sectors. At each iteration, the GPU memory storing visibilities is allocated and deallocated, causing the observed overhead. The number of CUDA calls increases with the number of sectors, which is equal to the number of MPI processes. With 2 GPUs these calls contribute to $\sim 20\%$ of the gridding time, while this number increases to $\sim 34\%$ in the 4 GPUs case. When large number of resources are used, it dominates the entire gridding time.

**FFT time.** While on the CPU the Fourier Transform is calculated using the FFTW, on the GPU the cuFFTMp library is adopted. In the *Small* test (Figure 5.1, top right panel), the speed-up shows a peculiar behaviour. Within a single node, the performance worsens from 1 to 2 GPUs. This is because the FFT accounts for two distinct contributions: plan creation and actual computation. Due to the small mesh size, the plan creation contribution is comparable to computation causing the measured per-

| | Nodes | MPI tasks (threads per task) | GPUs | Gridding (s) | Reduce (s) | FFT (s) | $w$-correction (s) | Total (s) |
|---|---|---|---|---|---|---|---|---|
| CPU tests | 32 | 1024 (1) | 0 | 4.5 | 9631.4 | 160.6 | 7.2 | 10246.0 |
| | 64 | 2048 (1) | 0 | 1.9 | 9598.2 | 107.1 | 3.5 | 10153.5 |
| | 128 | 4096 (1) | 0 | 1.1 | 9715.8 | 98.4 | 1.7 | 10266.5 |
| GPU tests | 32 | 128 (8) | 128 | 2.6 | 54.8 | 4.2 | 0.3 | 67.4 |
| | 64 | 256 (8) | 256 | 2.4 | 59.4 | 2.8 | 0.2 | 69.4 |
| | 128 | 512 (8) | 512 | 2.7 | 72.6 | 2.7 | 0.1 | 83.4 |

**Table 5.4:** *Strong scaling results for each step of the code for Large test, which has a mesh size of* $65,536 \times 65,536 \times 32$ *using LOFAR ILT dataset. Both full CPU and full GPU execution times in seconds are shown, with increasing computational resources for each. For these measurements we did not report the errors because multiple execution of this test would have consumed a large fraction of our available computing time.*

formance loss. From 2 to 4 GPUs the FFT scales linearly, exploiting the intra-node NVLink high-speed interconnect. On multiple nodes the FFT performance drops due to the progressively smaller problem size per GPU and the heavier communication overhead associated with the all-to-all required by the Fast Fourier Transform. Similarly, in the *Intermediate* case (Figure 5.1, bottom right panel) the FFT scales efficiently up to 8 nodes, decreasing when a larger number of GPUs is increasingly used. In addition, the cuFFTMp has a startup time for initialising NVSHMEM and the 2D plan for the Fourier transform. This startup time ranges from 0.5 s to 1.5 s for the Intermediate test, increasing with the number of MPI tasks.

For the *Small* and *Intermediate* tests on the CPU (Figure 5.1, top left panel), the FFT is computed through the hybrid FFTW MPI+OpenMP approach, keeping 8 OpenMP threads for each combination of MPI tasks. This explains the lack of scaling from 1 to 2 MPI tasks, where communication arising from calling multiple tasks determines an increase in the FFT time. The scaling is instead linear from 2 to 16 MPI tasks. In the CPU *Intermediate* test (bottom left panel in Figure 5.1), the scaling is close to ideal up to 16 nodes. In Figure 7 of (Gheller et al., 2023), it turns out that in the pure MPI case the FFTW scaling reproduces the expected linear scaling up to $\sim 10^2$ MPI tasks. Here the hybrid MPI+OpenMP FFTW is used instead. This choice is due to the fact that we have less GPUs than CPU cores in each node, and to use efficiently the entire node OpenMP parallelization is unavoidable. Hybrid FFTW is faster than the MPI one when less cores than available are utilised, but at least in these tests that we have performed, the code scalability gets worse with 128 MPI tasks distributed among 32 nodes. Understanding this scaling of the FFTW with OpenMP requires a deep code's profiling and it is beyond the scope of this thesis.

For the *Large* case, in pure CPU tests the FFT's performance improves by $\sim 60\%$ when passing from 32 to 64 nodes, while the performance gain is poor when passing from 64 to 128 nodes. With such a large number of MPI tasks, all to all communication dominates the runtime and the performance gain in the computation is almost completely suppressed. In the GPU tests, the FFT's performance improves by $\sim 50\%$ when passing from 32 to 64 nodes, while the performance gain drops when using 128 nodes. Again, this is due to the all-to-all communications and, at the same time, the progres-

sively lower computation related to an increasingly larger amount of MPI processes.
*w*-**correction time** Strong scaling tests of the *w*-correction reveal that the code scales almost linearly both for the *Small*, hybrid MPI+OpenMP, CPU tests and for the *Intermediate*, pure MPI, CPU tests. The same holds for GPUs, for which this step is implemented within a CUDA kernel. For this algorithm GPUs are more than order of magnitude faster than CPUs. In the *Large* tests, *w*-correction time keeps scaling linearly in both CPU and GPU tests. In particular, this kernel turns out to run efficiently on GPUs, there existing a factor of $\sim 20$ on average in runtime between CPU and GPU tests. *w*-correction is performed directly on the grid which has already been mapped on GPUs, such that no unnecessary memory management is needed.
**Full code** Overall, RICK shows a sub-linear scalability in both the *Small* and *Intermediate* configurations, both for the CPU and the GPU tests. As already pointed out above, such behaviour can be mainly ascribed to the impact on the code of the reduce related overhead and is also evident from the figures presented in Table 5.4. Additional factors, like the progressively smaller amount of data to process as the number of computing units grows, affect in particular the GPU runs.

### 5.3.3 Weak scaling tests

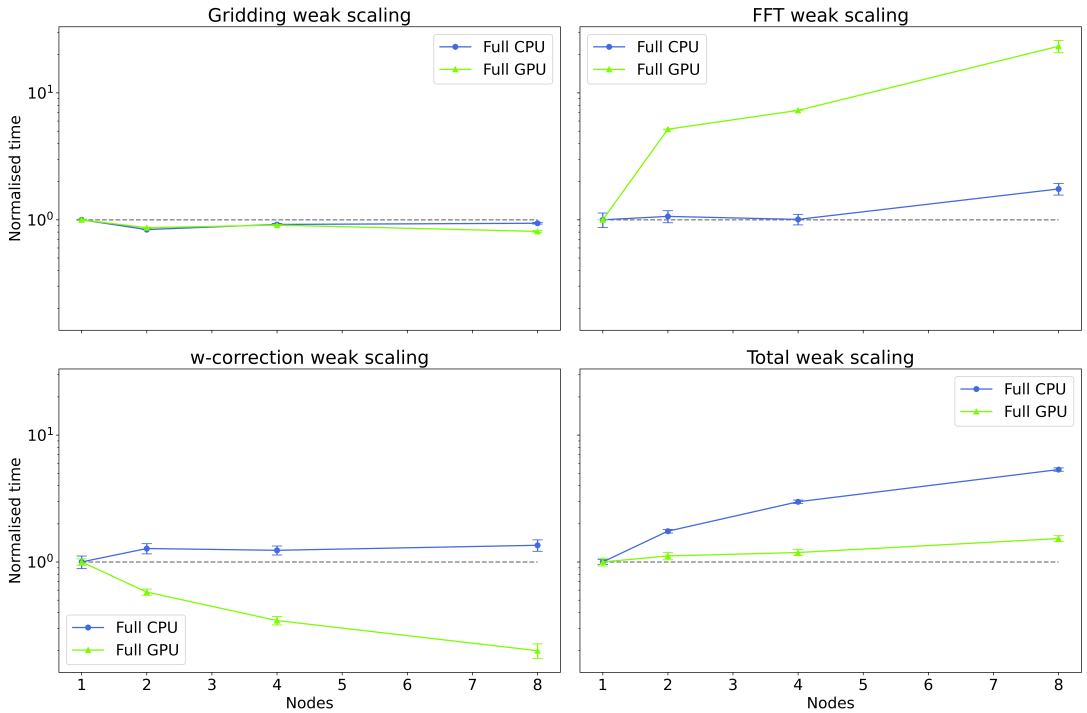



**Figure 5.2:** *Results for the weak scaling tests for each step of the code. From the top left corner, clockwise, results are shown for the gridding, FFT, w-correction, and total. The full CPU execution is shown in blue, while in green the full GPU enabling of the code. The horizontal dashed grey line represents the ideal weak scaling of the code. For each run 8 OpenMP threads were spawned.*

Weak scaling tests have been performed using the LOFAR Dutch dataset over 1, 2, 4 and 8 computing nodes, instancing 4 MPI tasks per node and doubling at each

step both the input size data and the number of $w$ planes. For each configuration, 8 OpenMP threads per MPI task have been spawned. In Figure 5.2, we present the runtime normalised to that of a single node (adopted as reference) of the different parts of the code and of the full code. Ideal weak scaling would result in a value of 1 regardless of the number of nodes.

Gridding, top-left panel, shows an almost ideal weak scalability for both the CPU and the GPU tests, with performance slightly improving increasing the number of nodes from 1 to 2 and showing small fluctuations for 4 and 8 nodes. The FFT weak scaling is presented in the top right panel. The hybrid MPI+OpenMP FFTW3 scales almost ideally from 1 to 4 nodes. On 8 nodes, the performance loss can be attributed to the impact of the all-to-all communication. For the GPU, cuFFTMp based tests, a significant performance drop is observed when the number of nodes increases from 1 to 2. This is mainly due to the switch from the intra-node high-speed NVlink interconnect to the lower bandwidth Infiniband network connecting different nodes. From 2 to 8 nodes, the performance is also worsening. This can be explained as a result of the way the weak scaling test is designed, maintaining constant the grid dimension and doubling at each step the number of $w$-planes. This increases the number of FFTs performed by the algorithm. However, the amount of computation per FFT that is performed by each GPU decreases with increasing the number of GPUs, leading to a computing efficiency loss, reflected by the weak scaling curve.

In the bottom-left panel, the weak scaling of the $w$-correction term is presented. The weak scaling for the CPU is nearly optimal, whereas in the case of the GPU, performance tends to increase significantly as the number of nodes grows. This behaviour can be interpreted as a consequence of the decreasing volume of data transferred from the GPU to the CPU as the number of GPUs increases. Specifically, when the number of GPUs is doubled, the portion of the image that needs to be copied back to the host halves. This has a positive impact on the time needed to transfer data between the device and the host.

When considering the overall scalability of the code, GPUs exhibit just a small reduction in efficiency as the number of nodes increases. In the case of the CPU instead, the deviation from the optimal scaling is evident. Once more, the efficiency loss is mainly due to the effect of the reduce, as we will discuss in the next section.

## 5.4 Discussion

The primary goal of the RICK code is to efficiently process huge datasets and generate large images in a reasonable time scale, of the order of seconds or minutes. In the tests provided, we utilised datasets with visibilities of up to 533 GB in size. Nevertheless, datasets of any size can be easily managed by splitting them into frequencies or time chunks. Chunks can be loaded sequentially, as described in (Gheller et al., 2024). This also enables to reserve an appropriate fraction of memory for the computational mesh.

By employing an appropriate number of processing units, we have successfully

generated images with a resolution of $65,536^2$ pixels. Utilising GPUs significantly enhances the performance of all code components, resulting in a considerable decrease in the time to solution. In particular, the exploitation of the cuFFTMp library reduces the impact of the Fourier Transform step. The corresponding computing time gain can range from one to two orders of magnitude compared to the FFTW CPU-only approach, depending on the scale of the problem. The current implementation of the library is highly efficient within a single node, taking advantage of the NVlink interconnect. However, its scalability is limited when multiple nodes are used due to the slower network and the use of the NVSHMEM protocol. In the case of the FFT, this protocol does not seem to offer optimal scaling on a large number of GPUs, unlike other solutions such as NCCL, which is used for the gridding part. In all the CPU tests, the hybrid MPI+OpenMP version of the FFTW ensures good scalability. However, a slight performance loss is found in a few configurations. This cannot be easily explained, but it is interesting noting that this issue has not been observed using the MPI-only version of the library.

Our tests have demonstrated that, when the problem size and computational setup are increased, the code's performance and scalability are significantly impacted by MPI communication. In addition to the parallel FFT, communication is performed by the reduce operations required to collect and add up the mesh data from all computing units during the gridding step. This last contribution tends to to dominate increasing the number of computing units. The remainder of this Section discusses the details of this aspect.

Visibilities are read evenly in parallel from an input data file in a time-log order, i.e. if we are dealing with an $8$ hours observation and $N = 8$, each MPI task reads data for one hour of observation. Conversely, the Cartesian computational mesh, where visibilities are gridded, is divided into rectangular slabs in the $u$-$v$ plane. Each slab is allocated to a distinct MPI task. Therefore, it is necessary to convert time-ordered into space-ordered data. To achieve this, the code iterates through the slabs. During each iteration $i$, each MPI task simultaneously computes its contribution to the $i$-th slab. Then, each MPI task stores its local contribution in an auxiliary buffer. Once local contributions are computed, they are summed together on a *target* buffer located on the $i$-th MPI task. Data collection and aggregation are achieved using an MPI reduction operation, which combines both communication (collecting data from several sources) and computation (calculating the sum of all contributions). As discussed in (Gheller et al., 2023), for large images the code spends $80 - 90\%$ of the total runtime in the reduce operation, when the reduce operation was implemented with standard Ireduce from OpenMPI library.

The overhead introduced in the code by the reduction operation depends on several factors, namely: *i*) the amount of data communicated among the MPI tasks; *ii*) the number of MPI reduce calls *iii*); the amount of computation required to perform the sum; *iv*) the network topology. These factors are interconnected. During our iterative procedure, the number of iterations over slabs is equal to $N$. Each MPI task performs

a reduce at every iteration, hence the theoretical reduce time $T_{\mathrm{R}}$ can be estimated as the sum of the communication ($T_{comm}$) and the calculation ($T_{comp}$) times. For the CPU implementation, using the OpenMPI library, $T_{\mathrm{R}}$ can be calculated as in Equation 2.1:

$$\begin{aligned} T_{Reduce} &= (T_{comm} + T_{comp}) \times N \\ &= \left(\beta \frac{D}{N} + \lambda \log N\right) \times N + \left(\frac{D}{N} t_{sum}\right) \times N \\ &= (\beta + t_{sum})D + \lambda N \log N \end{aligned} \tag{5.2}$$

Where $D$ is the total size of the data, $\beta$ quantifies the bandwidth and the network topology, $\lambda$ is the network latency, and $t_{\mathrm{sum}}$ is the time required for a single summation operation. The logarithmic term is due to the logarithmic tree algorithm used by the OpenMPI reduce.

The reduce time is then composed by two terms: the first depends on the data size and on $\beta$, while the second on the number of MPI processes. The latency term $\lambda$ is generally very low ($\leq 0.6\ \mu$s). In the strong scaling case, in which $D$ is maintained constant increasing the number of MPI tasks, the first term is dominant on the second because of the low latency: for this reason, we observe a constant reduce time (Figure 5.3, top panel). Only once we use a considerably large $N$, e.g. over 128 MPI tasks (as reported in the top panel of Figure 5.3), we start to observe an increase in the reduce time: however, such increase is not due to the processes themselves, but rather to the network topology, because of the non-ideal interconnection between the nodes and to the impact of the latency term. For the weak scaling, instead, we have $D$ that increases as well as $N$. The first term is still dominating, and for this reason we observe a reduce time which increases linearly with the data size (Figure 5.3, bottom panel).

The GPU implementation exploits the NCCL library, that implements a ring algorithm for the reduce: this means that $T_{\mathrm{R}}$ can be written as in Equation 2.3:

$$T_{Reduce} = (\beta + t_{sum})D + \lambda N^2 \tag{5.3}$$

Due to the quadratic dependency from $N$, latency becomes non-negligible using a smaller amount of processes. For example, OpenMPI would necessitate 100,000 MPI tasks to produce a latency of 1 second, whereas the NCCL would require only 1,000 MPI tasks. On the other hand, GPUs are capable of completing the sum operation much faster than the CPUs, because of their large computational throughput, about two order of magnitude larger than that of the CPU. This results in a considerably smaller $t_{\mathrm{sum}}$, that contributes in reducing the NCCL reduce time. In addition the NCCL reduce performs better than the MPI one due to *i*) the high-speed NVlink interconnection for intra-node GPU-GPU communication, *ii*) the number of network interface cards (NIC) that equals the number of accelerators per node, leading to a bandwidth 4 times bigger than pure CPU cases. Overall we see that the the reduce operation is about 20 times faster on the GPU for the *Small* test (see Table 5.3) and up to $\times 175$ faster

for the *Large* tests (see Table 5.4).

Table 5.4 also shows that the MPI Ireduce time remains constant scaling from 32 to 64 to 128 computing nodes. This has a progressively greater impact on the code compared to the more computationally intensive components of the algorithm. The GPU reduce time tends even to increase with the number of nodes. However, it is two orders of magnitude lower than that of the CPU. In both scenarios, increasing the number of nodes does not result in any improvement in speed, and in the case of the GPU, it actually leads to a decrease in performance.

However, it is extensively noted in Table 5.4 that it's possible to impact on the reduce algorithm and implementation (see Chapter 2 and Chapter 3), but it's not possible to impact on the code scalability, since MPI Reduce in this specific code involves an all-to-all, and the best we can achieve is a flat behaviour by increasing the number of processes. When $N$ is low, computation-to-communication ratio is higher and the code is still compute-bound, this leads to the scaling observed in *Small* and *Intermediate* tests, but when $N$ becomes high, and this is the case when many computing nodes are to be used due to large problems' memory requests, there is a turn-around point at which computation-to-communication ratio becomes lower than one, and in this case the best that can be achieved is the reduce constancy, from the theoretical point of view. We investigated several strategies to avoid the reduce utilization, like the possibility to distribute visibilities in space-order among the MPI processes before the gridding operation, when data are still point-like. However, without a grid, it becomes essential to introduce a sorting algorithm which arranges visibility according to their $v$ coordinate. After all, MPI processes need data exchanges involving again an MPI all-to-all communication, since data pertaining to each task will be spread among all the others. A first check led us to abandon this idea since the overhead of the sorting algorithm plus the the MPI all-to-all communication is greater than the MPI Reduce on the whole grid.

In general, in all test regimes the code runtime is strongly impacted by the reduce operation. In order to exploit with the maximum efficiency the available computing systems, it is crucial to select an hardware configuration that has the fewest computing units necessary to meet the memory needs. This also results in saving energy, since the problem is solved in approximately the same amount of time by while using less computational resources.

## 5.5 Conclusions

In this chapter we studied the RICK scalability in both intra-node and inter-node tests.

Our main achievements can be summarised as follows.

- Starting from the code presented in (Gheller et al., 2023), we managed to complete the full GPU porting of the code, including the FFT operation and the communication. Currently the code is capable of fully running on NVIDIA GPUs:

this minimises data movement between CPU and GPU and avoids overheads related to the CPU communication, exploiting the full potential of the GPU enabling.

- For the reduce operation we present an implementation for GPUs based on the NVIDIA NCCL library, which uses, where possible, high-speed interconnectivity. Moreover, we introduced an hybrid approach MPI+OpenMP to the reduce problem which has been used here for testing but will be detailed in a future paper.
- The FFT is instead accelerated using the cuFFTMp library of the NVIDIA HPC-SDK toolkit: with this the FFT problem can be distributed among multiple GPUs, which is critical considering the large volume of data processed compared to the memory of the devices. To exploit the full CPU computational capacity of a node we also tested a hybrid FFTW, which combines MPI tasks with OpenMP threads, which has a better scalability compared to the sole MPI FFTW thanks to a reduce communication surface.
- We tested performance and scalability of the code on Leonardo HPC cluster at CINECA (Italy), with tests on strong and weak scaling of every step that constitute our code using real LOFAR data (also comprehending International stations). Comparing our full-GPU code with the full-CPU, we observe a gain in total runtime of a factor $\times 9$ for the *Small* and *Intermediate* dataset configuration within a single node, and up to $\sim \times 130$ for the *Large* configuration over a large number of nodes.
- Thanks to GPU offloading, our code's computational costs have been greatly reduced. The runtime is now mainly affected by communication. To fully leverage the improved capabilities of this implementation, it is essential to select a computational setup that match the problem size and and minimises communication as much as possible.

Overall, RICK stands out as an innovative radio imaging software that fully utilises GPUs, making it a promising solution for the future software suites for processing big radio astronomical data, such as those expected for the SKA. Future advancements will focus on optimising communication and providing full support for parallel I/O in radio astronomy measurement sets.

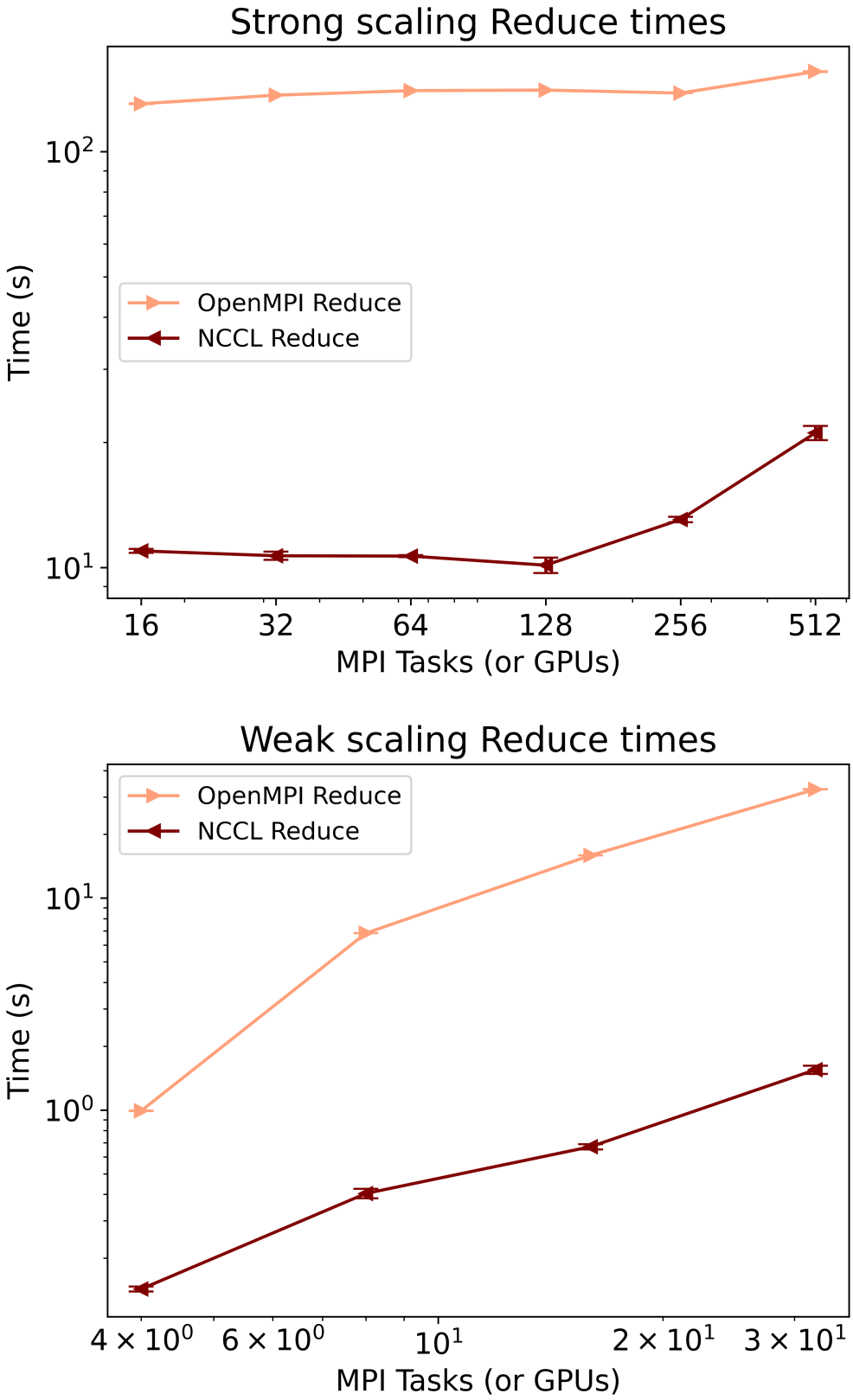



**Figure 5.3:** *Time results for the reduce operation versus the number of MPI tasks or GPUs for the Intermediate strong scaling test (top panel), and the weak scaling test (bottom panel). Comparison between the OpenMPI Ireduce (in pink) and NCCL GPU reduce (in brown).*

*This page intentionally left blank.*

# 6

# RICK ENERGY EFFICIENCY TESTS

## 6.1 Introduction

From the point of view of energy-to-solution, the task of code's efficiency becomes more challenging by considering that current pre-exascale HPC systems consume tens of $MW$, running under full workload, and it is easy to understand that this trend will become unsustainable in the near future when several exascale machines will be available. The energy efficiency of HPC platforms is measured as energy efficiency = $GFlops/W$, i.e. billions of floating point operations per watt, and is classified in the Green500[1] list. In this chapter we study various CPU only and accelerated solutions in order to determine optimal configurations in terms of both time-to-solution and energy-to-solution for radio astronomy.
The chapter is organized as follows: in Section 6.2 we will discuss the code's implementations, in Section 6.3 we will define the **green productivity** and its relevance for our code. Experimental setup will be introduced and in Section 6.4. Tests performed in single-node will be presented and discussed in Section 6.5, whereas Section 6.6 will be devoted to multi-node tests. Conclusions will be drawn in Section 6.7.

## 6.2 Code implementations

The code is written in C, with C++ extensions just to add the CUDA/HIP support for GPUs. One of the goals of this code is the portability, in order to run on different architectures. In this section we will discuss the three implementations that have been tested in this work. We have a pure MPI implementation, an hybrid MPI+OpenMP implementation and a MPI+CUDA/HIP implementation to use GPUs.
As shown in Figure 2.9 RICK performs the five algorithmic steps described above and can be run both in serial and in parallel. Of course the scale of SKA imaging necessitates massively parallel systems, so hereafter we will only consider the parallel implementations of the code.
Reading data from the filesystem is done in parallel, but the current implementation

[1] `https://top500.org/lists/green500/list/2024/06/`

has also the possibility to handle datasets of any size even on a single node, because they can be easily managed by splitting stacked visibility data, i.e. large input data can contain visibilities from observations at different frequencies, into frequencies or time chunks. Instead of loading all the data at once, data coming from different frequencies are assigned to a chunk. Chunks can be loaded sequentially, as described in (Gheller et al., 2024). This also enables to reserve an appropriate fraction of memory for the computational mesh.

Data are initially distributed in time-log order, i.e. if input data come from 8 hours of observations and 8 MPI tasks are used, task 0 will read the first hour of observation, task 1 will read the second hour and so on. This does not require that each observation actually needs to be of the same length but interpretation of the time stacking does help if each observation is roughly the same. In order to reconstruct the sky image, each MPI task processes single sectors in the u-v plane, necessitating a transformation from time-log order to space-log order. In time-log order input, each task has visibilities pertaining to all the other ones, and MPI communication is unavoidable. When visibilities in each sector are gridded, an MPI Reduce operation is performed on the grid with target the MPI task that owns the sector.

### 6.2.1 MPI implementation

In the MPI implementation one assigns an MPI task to each of the available CPU cores. This means that the five algorithmic steps performed by the code are fully distributed among all the MPI processes. There are as many sectors as MPI tasks and consequently the same number of reduce operation is needed, leading to a communication overhead (see (Gheller et al., 2023) and (De Rubeis et al., 2024)).
Communication has been largely discussed in Section 5.4 and in Section 5 of (De Rubeis et al., 2024), so a detailed treatment is beyond the scope of this chapter. In this pure MPI case, we've used the standard MPI Reduce, available in every MPI implementation.

### 6.2.2 Hybrid MPI/OpenMP implementation

To mitigate the impact of MPI communication, it became essential to include in our code a hybrid parallelization with MPI and OpenMP. The hybrid implementation reduces the communication impact by diminishing the number of MPI Reduce calls and reduces the communication surface because MPI Reduce involves less MPI processes. Furthermore, we have designed a hybrid technique combining MPI and OpenMP and exploiting Non-Uniform Memory Access (NUMA) topology. It works as follows: as first, two MPI communicators are built:

1. An intra-node communicator, where only the MPI ranks pertaining to each computing node are included and are ranked from 0 to $P-1$, with $P$ being the number of MPI ranks on every node. When multi-socket nodes or when multi-NUMA

regions CPUs are available, MPI processes are automatically assigned to each NUMA region in a round-robin fashion.
2. An inter-node communicator that groups all ranks 0 (which are the node masters) in the intra-node communicators.

In communicator 1, every rank knows about all other ranks in a given communicator so rank $i$ will communicate to rank $i + 1$ in a ring with $P \rightarrow 0$. Each MPI task spawns two threads. Thread 1 is in charge of calling the ring shared-memory reduce function involving all the other tasks in the intra-node communicator, while thread 0 manages the MPI communications among the nodes. It is important to say that although each task needs at least two threads, only task 0 is actually involved in inter-node MPI communications. So in the current implementation all the other tasks spawn *idle* threads. The reduce operation in one node is in shared-memory because MPI shared windows are used to create direct memory access channels between the MPI tasks in the same communicator. When the shared-memory reduce is done, task 0 (i.e the master) gathers the result: this is the total summation in case of a single node run, and it is a partial summation when there are multi-nodes. In this work hybrid tests have been performed on just one node. How the reduce is handled in multi-node cases is discussed in Section 5.2.3 and in Section 3.3 of (De Rubeis et al., 2024).

### 6.2.3 GPU implementation

To fully exploit the computing power of accelerators we have designed a GPU implementation of the code which performs three main steps of the algorithm directly on the accelerators: gridding, reduce, $w$-correction. Grid is allocated on the GPU memory at the beginning, whereas visibilities are loaded on the device in each sector, then the gridding function is turned to a gridding kernel which maps visibilities on the 2D mesh. After this step, the gridded data are entirely present in the GPU memory, such that direct GPU-GPU communication can be performed without transferring data back to the host. GPU-GPU communication is performed on AMD GPUs with the ROCm Collective Communication Library (RCCL[2]), which implements the reduce operation as a ring intra-node, and an inter-node ring, where GPUs assigned to the MPI tasks communicate with Remote Direct Memory Access (RDMA) with GPUs in different nodes without passing through the CPUs (A. Li et al., 2019; Sensi et al., 2024). Fast Fourier Transform is still computed on the CPU because there is no available implementation for AMD GPUs yet.

## 6.3 Green productivity

The goal of this work is to find the best trade-off in the code's energy efficiency and performance when *more computing resources/different configurations* are used. However

[2] `https://rocm.docs.amd.com/projects/rccl/en/latest/`

we considered that although is easy to measure either when a code is faster or less energy consuming, it is not trivial to find a physical quantity that relates the energy consumption with the runtime in a meaningful way. This led us to the introduction of the **green productivity**. It has the following definition:

$$GP = \frac{T_0/T_N}{\alpha E_N/E_0} \tag{6.1}$$

Here $T_0$ and $E_0$ are runtime and energy consumption of a reference configuration, respectively. Clearly, when the same code implementation is being tested and the only thing changing is the number of computing resources, the quantity at the numerator is simply the speedup in a strong/weak scaling test. $\alpha$ is a weight factor which changes depending on what we consider more important among performance and energy consumption. In this work we treated them with the same importance and chose $\alpha = 1$.

## 6.4 Experimental setup

In this Section we present and discuss the different tests with the corresponding results. We focused on three different code's implementations, i.e. pure MPI, hybrid MPI+OpenMP and MPI+HIP. We relied on the SLURM (Simple Linux Utility for Resource Management) energy counters for the energy measurements in both CPU only and CPU+GPU tests.
With these counters, the energy consumption of the entire job is measured, apart from I/O and memory accesses, even though the energy consumed is still impacted by time spent doing IO. Here we have focused on the total energy consumption of the whole code. Other libraries, like PAPI[3], permit the energy profiling of codes' functions with internal calls.

Input data in our tests come from 8 hour observations from the LOFAR HBA Inner Station at different frequency channels, with each channel using $\sim 4.4GB$ of storage. We stacked two out of these datasets for the *Single-node* tests and 18 of them for the *Multi-node* tests. Details on memory occupancy are shown in Table 6.1.
The tests have been run on the Setonix-CPU and Setonix-GPU machines available at the Pawsey Supercomputing Research Centre (PSC) in Perth (WA). Setonix-GPU is ranked as $28^{th}$ in the June 2024 Top500 list and $10^{th}$ in the June 2024 Green500 list.
CPU partition is made of 1088 computing nodes equipped with 2 × AMD EPYC 7763 "Milan" 64 cores; GPU partition is made of 154 computing nodes equipped with 1 × AMD optimised 3rd Gen EPYC "Trento" 64 cores and 8 GCDs (from 4x "AMD MI250X" cards, each card with 2 GCDs), 128 GB HBM2e. CPU-GPU and GPU-GPU interconnections inside each node are guaranteed by the InfinityFabric technology. For node-node connections, both partitions have Slingshot interconnections. In all the

[3] `https://github.com/icl-utk-edu/papi`

| | Single-node | Multi-node |
|---|---|---|
| N. visibilities (approx) | $1.08\times10^9$ | $9.72\times10^9$ |
| Input data size (GB) | 8.8 | 78 |
| $N_u \times N_v \times N_w$ | $4096^2\times64$ | $16384^2 \times 24$ |
| Mesh size (GB) | 43.24 | 194.11 |

**Table 6.1:** *Configuration and computational mesh used in the Single-node and Multi-node tests. The mesh size takes into account the total amount of memory required for the real and imaginary part depending on the size of the grid.*

tests the code has been compiled with **clang-16**[4] and **MPICH**[5] implementation has been utilized.

## 6.5 Single-node tests

Here we show the results for several CPU hybrid MPI+OpenMP configurations and the CPU+GPU configuration referred to the pure MPI run. As we mentioned in Section 2, the pure MPI tests utilize the standard MPI Reduce function deployed by the MPI library, whereas MPI+OpenMP tests utilize our hybrid reduce implementation and GPU tests utilize RCCL reduce implementation. In Table 6.2, we show the results for the pure MPI test, the best configuration for the hybrid MPI+OpenMP test and CPU+GPU test. Results are averages and standard deviations over four runs for each configuration. The hybrid implementation is $4$ times faster and $3.3$ times more green than the pure MPI run, while GPU implementation is $8.2$ times faster and $3$ times more green than pure MPI. The main difference is in the reduce operation, which is $20$ and $40$ times faster for the hybrid and GPU implementation compared to MPI, respectively. The performance difference in FFTW in GPU runs is due to the fact that it's still on the CPU and no OpenMP threadization is active for this test. Pure MPI tests are the fastest in gridding time, even when comparing them to GPU tests. This behaviour is due to the overhead related to HIP GPU memory management, and, more specifically, to repeated *hipMalloc* and *hipFree* calls that are implied by the iteration procedure through mesh sectors, as discussed in (De Rubeis et al., 2024). However, in order to understand which configuration is actually the most efficient we show in Figure 6.1 the green productivity, where $T_0$ and $E_0$ in Equation 6.1 are $T_{MPI}$ and $E_{MPI}$. We notice that the hybrid implementation with 16 MPI tasks spawning 8 OpenMP threads each and the CPU+GPU configuration turn out to have a similar green productivity.

We can make the following considerations: 1) when the green productivity is almost the same it's up to the user whether to choose the fastest or the most green configuration; 2) it is important to add that, at least at the Setonix machine, asking for GPUs requires $8$ times more core hours. For this reason, when a problem does fit in a single node, we found that the best solution is the hybrid one.

[4] `https://releases.llvm.org/16.0.0/tools/clang/docs/ReleaseNotes.html`
[5] `https://docs.nersc.gov/development/programming-models/mpi/cray-mpich/`

| | MPI | Best MPI/OMP | GPU |
|---|---|---|---|
| Energy (KJ) | 60.8375 ± 2.6709 | 18.3325 ± 0.2175 | 20.2325 ± 0.3790 |
| Total time (sec) | 95.9533 ± 0.1560 | 24.4133 ± 1.7793 | 11.7309 ± 0.0446 |
| Gridding time (sec) | 0.6624 ± 0.0013 | 1.5677 ± 0.0183 | 1.0925 ± 0.0001 |
| Reduce time (sec) | 84.3082 ± 0.1399 | 4.3494 ± 0.0054 | 2.0065 ± 0.0118 |
| FFTW time (sec) | 1.1941 ± 0.0982 | 1.8458 ± 0.0561 | 4.8540 ± 0.0205 |
| $w$-corr. time (sec) | 0.4039 ± 0.0007 | 0.5201 ± 0.0073 | 0.1112 ± 0.0003 |

**Table 6.2:** *Total energy and runtimes for the relevant code portions in the MPI, MPI+OpenMP and GPU cases respectively. Here we report averages and standard deviations over four runs each.*

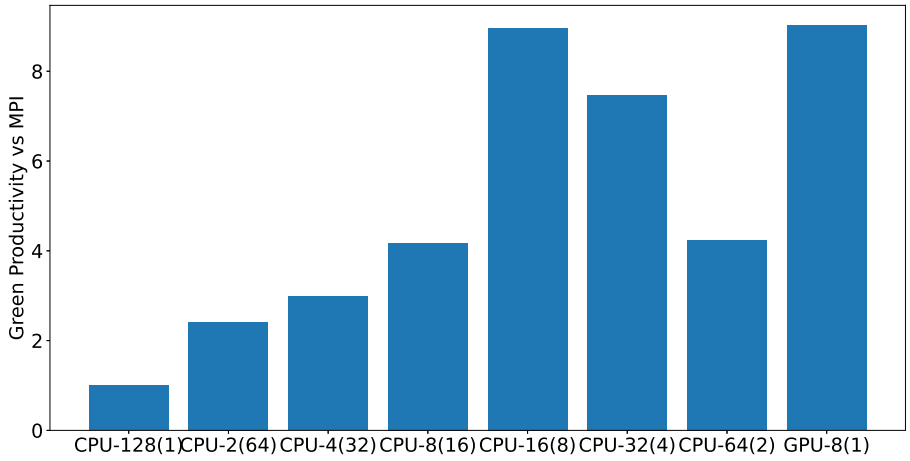



**Figure 6.1:** *Green productivity referred to the pure MPI run of the different hybrid MPI+OpenMP configurations and CPU+GPU.*

## 6.6 Multi-node tests

This section will be devoted to the tests with the large input data and grid size involving many computing nodes. These are strong scalability tests, to understand the memory imprint of the code compared to the computational part. We will investigate energy-to-solution and time-to-solution of pure CPU and CPU+GPU configuration. In the end we will study the green productivity referred to the lowest resources solution for each configuration.

### 6.6.1 CPU tests

Here we discuss the results for the CPU tests. We have chosen a combination of input data and grid size that doesn't fit in a single node, in order to include network traffic related to IO operations using the Lustre filesystem, which does impact on the reduce operation. We have done a strong scalability test with 2 nodes (256 MPI tasks), 4 nodes (512 MPI tasks), 8 nodes (1024 MPI tasks), 16 nodes (2048 MPI tasks), 32 nodes (4096 MPI tasks). We argued that multiple reduce calls are required by the code, especially

in pure MPI runs. This means that the code is expected to be strongly memory-bound. To study the memory imprint and the energy-to-solution in the code, we have run RICK under the same dataset and grid size configurations, but modifying at each step the CPU frequency. In particular, Setonix-CPU allows to select:

- **Default frequency**: The frequency is established by the OS depending on the computing demand of the specific code function. It changes during the code execution but profiling reveals that it remains close to the highest CPU frequency achievable.
- **High frequency**: The highest frequency achievable from the CPU, which is $2.60GHz$.
- **Medium frequency**: The CPU frequency is set to $2.00GHz$.
- **Low frequency**: The CPU frequency is set to $1.50GHz$.

We show in Figure 6.2 the fraction of runtime spent in the reduce operation as a function of the number of computing nodes, by changing the CPU frequency. We notice that this fraction increases with the number of nodes, eventually saturating around $\sim 95 - 96\%$ in all the configurations. In this case, when the reduce impact dominates the runtime, it makes sense to diminish the CPU frequency because the actual computational part is almost negligible. In this test, the pure MPI configuration has been used, and no MPI+OpenMP test is available. Indeed, the hybrid code implementation would have been diminished the communication surface and, as a consequence, the reduce runtime fraction in Figure 6.2.

The lack of the MPI+OpenMP test is due to the following shortcomings:

- **Problem size and buffer limitations**: The hybrid reduce relies on the MPI Ireduce, i.e. the non-blocking MPI Reduce function, for node-node communications through the network. However, partial results are collected by node's master, which then performs the MPI communication with the other masters. If the problem size is too large the master rank cannot allocate such a huge buffer needed by the Ireduce implementation, resulting in a code crash. This has been a problem for the strong scalability test, especially when the number of computing nodes is low. Filling all the available cores with MPI allows each process to allocate smaller and smaller buffers when the problem size is fixed. Running the code with a small grid would have been outside the scope of this paper since no relevant scalability test could have been done and the arithmetic intensity of the code would have been reduced as well.
- **Lack of inter-node hybrid reduce**: The current implementation of the hybrid reduce is optimized for intra-node only. The implementation of a ring reduce among the masters of each node is under development, and avoids the bufferization problem discussed above by utilizing MPI shared windows and MPI Put/Get functions. The main difficulty happens again for large problem sizes, since the entire communication cannot be performed all at once and data need to be split in chunks to be shared one-by-one. This requires several MPI Win Flush

calls, which currently introduce bottlenecks and sometimes to code hangs. In a future work we plan to fix this bugs and perform MPI+OpenMP CPU tests for node-node configurations as well.

In the left panel of Figure 6.3 we show the fraction of energy savings when we compare the default, medium and low frequencies to the highest frequency, as a function of the computing resources. We notice that default and high frequencies have approximately the same energy consumption, with small oscillations which anyway do not exceed $5\%$. It is more interesting what happens when we set the medium and low frequencies, because we reach up an energy saving of $25\%$ and $30\%$ when compared to the highest frequency, respectively. The right panel of Figure 6.3 shows the performance degradation which does happen when we change the CPU frequency. When we set the default frequency, we notice that the code is slower by $2-4\%$ with some oscillations meaning that in certain code portions the OS sets up the default frequency to the maximum frequency. For the medium and low frequencies, we find a performance degradation of $4-5\%$ and $8-10\%$ on average, respectively. Thus we have around $5$ times more energy saving than performance degradation in percentage when we set the medium CPU frequency and around $3$ times when we set the lowest CPU frequency.

### 6.6.2 GPU tests

We have done tests with 4, 8 and 16 GPU nodes, equipped with 32, 64 and 128 accelerators, respectively. In the following tests gridding, reduce and $w$-correction have been done on GPUs, while the FFT is still performed with the FFTW on CPUs. In the left panel of Figure 6.4 we show the ratio between the energy consumed by the pure CPU tests and the energy consumed by the CPU+GPU tests, the latter being $6-7$ times more green than the former with high/default frequencies and $4-5$ times more green with medium/low frequencies. In the right panel of Figure 6.4 we show the ratio between the pure CPU tests' runtime and CPU+GPU tests' runtime, the latter being faster by a factor $9-11$ for high/default frequencies and a factor $10-12$ for medium/low frequencies. For these *Multi-node* tests GPUs turn out to be definitely the best choice because they're better in both energy saving and performance.

### 6.6.3 Green productivity

In Figure 6.5 we plot the green productivity of each specific configuration compared to its lowest node one, in this case $X_0 = X_2$ where $X$ can be both $T$ and $E$. Because the code is memory-bound, in strong scalability tests increasing the computing resources does not lead to a speedup. Sometimes the performance becomes worse because of the communication overhead due to the increase of reduce calls. The best configuration in the one with the highest green productivity, which in this case always corresponds to the lowest node configuration. However, in pure CPU cases green productivity

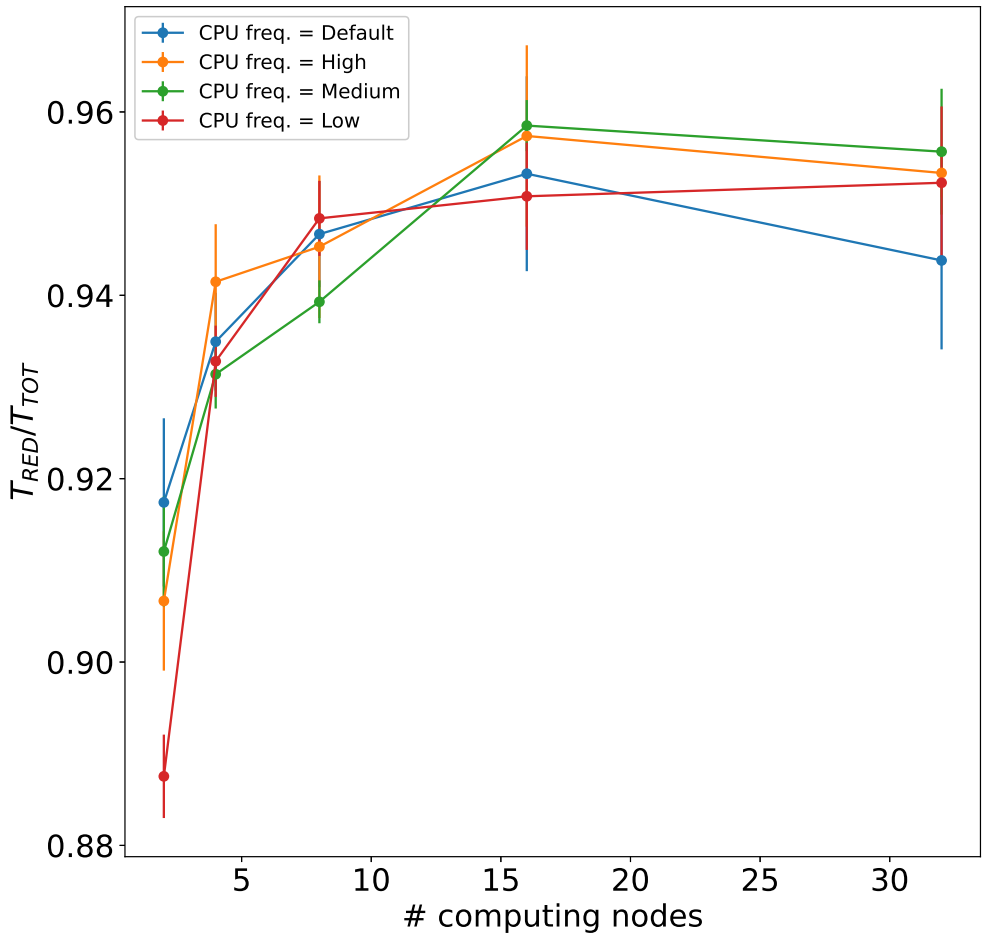



**Figure 6.2:** *Fraction of runtime spent in the reduce operation as a function of the number of computing nodes, for different CPU frequencies.*

drops much steeper even by moving from 2 to 4 nodes, whereas for GPUs it remains roughly constant when passing from 4 to 8 nodes. The sudden drop happening with 16 GPU nodes is explained by considering that when more and more accelerators are used by keeping the problem size constant, the gridding time stops scaling, again for overheads in HIP GPU memory management, as turned out in the discussion about *Single-node* tests.

## 6.7 Conclusions

The results discussed in the previous Section show that finding the configuration to achieve the best compromise between time-to-solution and energy-to-solution is not trivial. In RICK, it does depend on both data structures and the number of computing

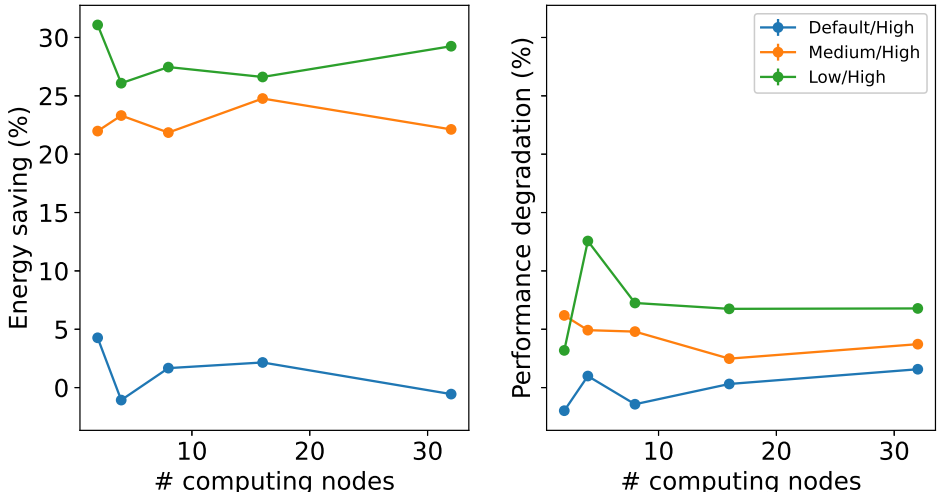



**Figure 6.3:** *Left: energy saving compared to the highest CPU frequency for default (blue), medium (orange), low (green) CPU frequencies as a function of computing nodes. Right: performance degradation of default, medium and low CPU frequencies compared to the highest CPU frequency as a function of computing nodes.*

resources used.

Our results can be summarized as follows:

- The usage of MPI and distributed GPU computing will be unavoidable when hundreds of petabytes of input data per year will be delivered, otherwise data processing in chunks is needed, raising large I/O overhead.
- The accelerated version of the code is always faster than the pure CPU version, but when a problem fits in one node it is not the greenest solution.
- For large input data and very high resolutions, as it will be for SKA, many computing nodes will be needed to perform radio imaging, and GPUs become exceptionally faster and more green than CPUs, in particular due to the high difference in MPI communication.
- Green productivity is a measure to relate code performance and energy efficiency, and fine-tuning with the weighting factor is useful in order to focus more on one out of them.

To increase both the energy-to-solution and time-to-solution for the GPU implementation, our next step will be the inclusion of the distributed FFT version for AMD GPUs, which will avoid unnecessary memory movements back and forth from device to host, after the reduce operation, and then from host to device to perform $w$-correction at the end.

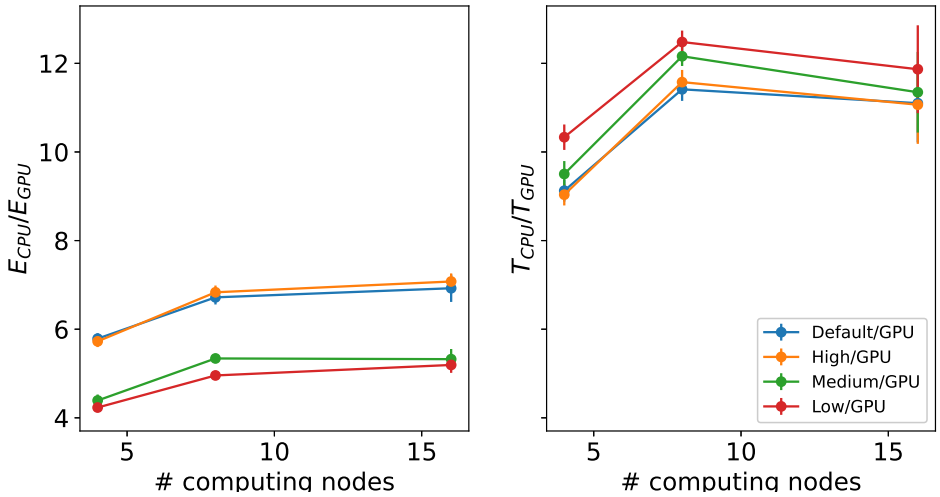



**Figure 6.4:** *Left: ratio between the energy in the pure CPU case and the energy in the CPU+GPU case, at different CPU frequencies, as a function of computing nodes. Right: ratio between the CPU runtime and CPU+GPU runtime, at different CPU frequencies, as a function of computing nodes. In the GPU tests both CPU and GPU frequencies are set by the OS to their default values.*

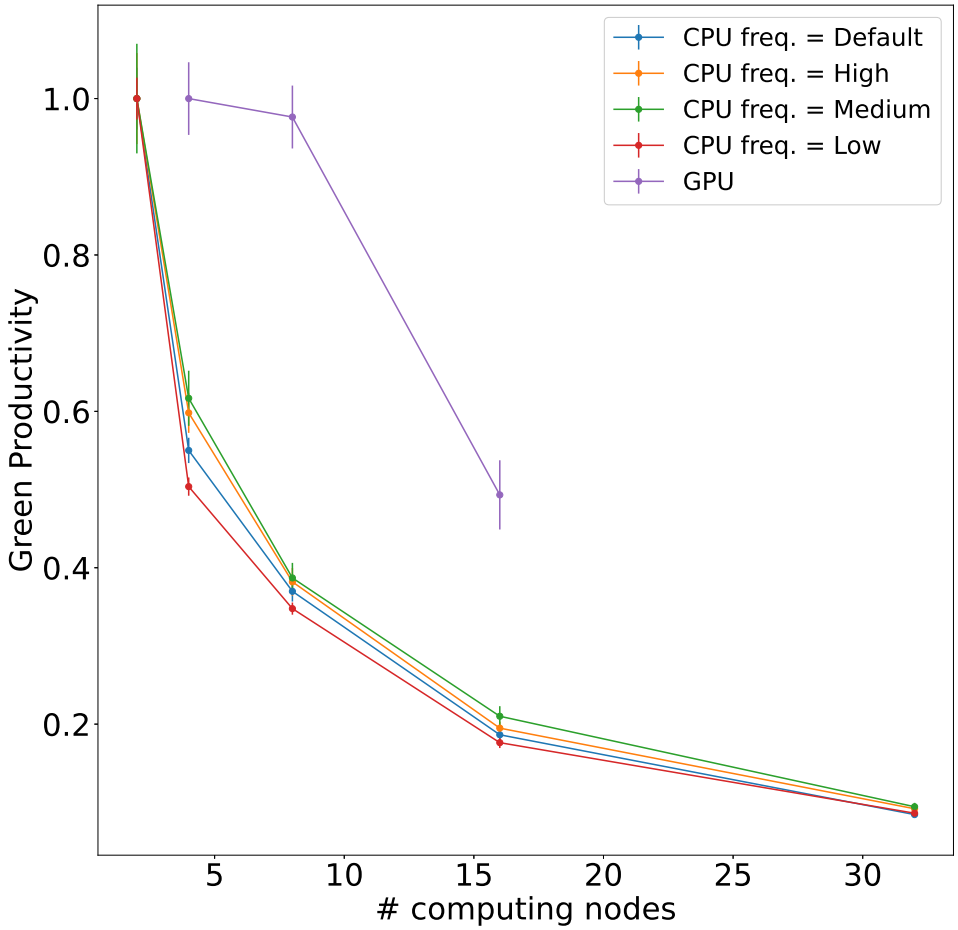



**Figure 6.5:** *Green productivity of CPU tests and CPU+GPU tests taking as reference the lowest node configuration for each different test, as a function of computing nodes.*

*This page intentionally left blank.*

# 7

# Conclusions

Green computing has now pivotal importance in every field of science, and in particular in astrophysics and cosmology, since always more complex simulations involving up to trillions of particles are currently being run on supercomputers, while at the same time new instruments have been built with the purpose to shed light on the Universe mysteries, like the nature of dark matter and dark energy. These instruments are going to produce up to petabytes data every day, which eventually need to be stored, analyzed and processed. This led naturally to the exploitation of supercomputers.

We found in the first part of this thesis that energy-to-solution may be lowered in two ways:

- Putting much effort to develop fast codes exploiting heterogeneous machines with parallel computing and GPUs. A single GPU consumes more than a CPU in idle state, however they're so much faster in several applications that, even if the power absorption is large, the total energy consumed is much smaller than CPU equivalents.
- The awareness about algorithmic imprint, discussed in Chapter 3, i.e. writing codes which utilize more efficient instructions.

The latter is the key point of the first part of the thesis, even if its effect is smaller than the runtime gain, I estimated its contribution around $\sim 10\%$ in the reduce operation implemented by my collaborators and I. This optimized version of the reduce, introduced and described in Section 3.2, is faster than the standard MPI solutions and utilizes more efficient instructions as well. The effect reaches $10\%$ with $12$ MPI tasks in an Intel machines, which is much less than runtime gain, but it's non-negligible. However, deeper analyses confirm that its effect increases with $N$. A future study will include new Intel and AMD machines with many cores, in order to grasp how the effect grows when $N$ becomes large. To understand which instructions are actually the cheapest, a deep profiling involving a study in what's going on at assembler level is unavoidable, and will be the main future perspective. This instruction profiling will reveal the most efficient instructions, which can be then ported in whatever scientific code and not only in the reduce operation. Effort on developing green algorithms will

focus the importance of green awareness in the future, which will not only be concentrated on hybrid or GPU parallelism. Considering to run the reduce on a machine consuming tens of $MW$ under full workload, without any code speedup it is in principle possible to save $1MW$ every $10MW$ even if the algorithmic imprint impacts "only" $10\%$.

In the second part of this thesis we have introduced our RICK code, and we have run it at the Leonardo supercomputer and at the Setonix machine. The two platforms have different architectures, the former being an Intel CPU + NVIDIA GPU machine and the latter being an AMD CPU + AMD GPU one.

Our results show that GPUs are always the fastest solution but sometimes they're not the greenest one. Indeed, we can summarize our tests as follows:

- OpenMP tests remove MPI overhead but force users to run the code entirely in one node, which is a disadvantage when large input data and grids are needed, as it happens for LOFAR and SKA data analysis. The code anyway offers the possibility to load datasets split in frequencies/observational time chunks (see Chapter 4) but leads to I/O overhead if these chunks become too many (see Section 1.9).
- Pure MPI tests solve the memory limitation issue but at the same time introduce large communication overhead, as stressed in Chapter 3 and Section 5.4. Especially for a strongly memory-bound code like RICK, by increasing the computing resources the code runtime will be entirely dominated by the communication (see Figure 6.2).
- Hybrid MPI+OpenMP tests are the most reasonable trade-off when GPUs are not available. Indeed, the communication impact is reduced, thanks also to the new implementation, which allows users to diminish the MPI tasks without losing any computing power, since the node is entirely filled with OpenMP threads.
- MPI+GPUs is the fastest solution in any configuration, as it has been shown in the results of Chapter 4, Chapter 5, Chapter 6. The difference, as we already discussed in Chapter 5,Chapter 6, is in the implementation of the Fast Fourier Transform. The current code version supports only the FFT running on NVIDIA GPUs, while in the AMD case explored in Chapter 6 the FFT is still run on the CPU, leading to memory transfers back and forth. However, it is shown in Figure 6.4 that GPU implementation is much faster and greener even without the FFT, when large computing resources are required by the problem.

Energy-to-solution has been the goal of this thesis, however a green code which is too slow in runtime is not useful for astronomers, such that we had to find a quantity to compromise energy-to-solution and time-to-solution, which we called **green productivity** (see Section 6.3). The relevant results are collected and discussed in Chapter 6. Thanks to the Pawsey Supercomputing Centre staff, we could exploit energy counters and fine-tune parameters like the CPU frequency, in order to study the impact on the total energy consumption. Here the take-home messages:

- When the problem fits in one node, i.e. non-VLBI observations, the GPU solution is again the fastest one but not the greenest one (see Table 6.2), and in terms of green productivity (Equation 6.1) it's equivalent to one specific hybrid MPI+OpenMP configuration (see Figure 6.1), but at the same time the GPU solution requires 8 times more core hours compared to the CPU solution, meaning that hybrid solution is the most efficient one in single-node cases.
- When many computing resources are needed to fit the problem memory requirements, i.e. LOFAR VLBI, when high resolution images are unavoidably needed, GPUs turn out to be by far the fastest and the greenest solution, as clearly shown in Figure 6.4. In terms of green productivity, in Figure 6.5 we took as reference the lowest node number configurations, and in both CPU and GPU solutions the most efficient one is the lowest resource configuration itself.
- Tuning the CPU frequency led to the interesting results discussed in Section 6.6.1 and shown in Figure 6.2 and Figure 6.3. The code turns out to be strongly memory-bound, such that diminishing the CPU frequency does not lead to a significant performance loss, while impacts more on the energy-to-solution. In this case it's up to the users whether to choose either the medium or the low frequency configuration.

RICK is still under development since from both scientific and algorithmic points of view improvements can be implemented. For instance, deconvolution (Starck et al., 2002; F. Li et al., 2011; Hardy, 2013; A. Offringa et al., 2017), needed to clean the image for incomplete sampling in Fourier space by the antennas, is not yet performed by the code, which produces the so called *dirty image*. Also, weighting and tapering (Yatawatta, 2014) are being improved.

For the computational part, the code can be optimized as follows:

- **Hybrid multi-node reduce** The current reduce implementation on CPU is able to speedup the intra-node communication without losing computing power thanks to the hybrid MPI+OpenMP communication. However, the implementation of a new algorithm for the inter-node reduce is necessary because currently the code, in multi-nodes, relies on the standard reduce available in the MPI library.
- **Distributed FFT for AMD GPUs** The Fast Fourier Transform is still a limiting factor in AMD machines, since a distributed version of rocFFT[1] is not yet available. This limits the full GPU computing power, because memory transfer from device to host is necessary between the reduce and the FFT and again from host to device between the FFT and the $w$-correction. A full GPU version with the FFT included would avoid this overhead in memory transfer, resulting in an improvement in both energy-to-solution and time-to-solution.
- **ADIOS2 inclusion** In Section 1.9 the I/O problem in radio astronomy has been introduced, arguing that it becomes a bottleneck especially when several datasets have to be loaded and processed one by one. However this is not the only case in

[1] `https://rocm.docs.amd.com/projects/rocFFT/en/latest/`

which I/O becomes a bottleneck. When parallel I/O is used, CPU's memory links with the file-system are usually much smaller than the number of available cores. If the implementation is not streaming-flow aware, memory contention can significantly limit the performance in reading data from the file-system. ADIOS2 will help us with its API to handle I/O in the proper way, in order to achieve the best performance.

- **MGARD and communication** The most challenging solution to be implemented is thought to reduce the communication impact, which has been discussed during the whole thesis and considered as "worst enemy", since, in particular when many computing nodes are used, turns out to dominate the entire runtime. MGARD, which is already implemented in ADIOS2, allows to perform data compression with or without losing any information. If the code were able to compress gridded visibilities between the gridding step and the reduce step, the communication would be done onto smaller data structures, eventually leading to a diminished communication impact.

This last optimization is in principle feasible, however it requires that the sum of compression step and reduce on a smaller grid is smaller than the reduce on the uncompressed grid. One possibility to speedup this code part is to rely on ADIOS2 streaming-flow, i.e. investigating the possibility to perform the compression in a non-blocking way while the task starts to perform the reduce operation. This is easier for the CPU code since it can trust the OpenMP parallelization, i.e. a group of threads is compressing the grid while the remaining ones are involved in the reduce. When GPUs will be used, this is even trickier, since other *cudaStreams*/*hipStreams* will be necessary to include this new task.

*This page intentionally left blank.*

# Bibliography


Akl, Selim G (1997). *Parallel computation: models and methods*. Prentice-Hall, Inc.

Alt, Lukas et al. (2024). "An Experimental Setup to Evaluate RAPL Energy Counters for Heterogeneous Memory". In: *Proceedings of the 15th ACM/SPEC International Conference on Performance Engineering*, pp. 71–82.

Amdahl, Gene M. (1967). "Validity of the single processor approach to achieving large scale computing capabilities". In: *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*. AFIPS '67 (Spring). Atlantic City, New Jersey: Association for Computing Machinery, pp. 483–485. ISBN: 9781450378956. DOI: 10.1145/1465482.1465560. URL: https://doi.org/10.1145/1465482.1465560.

Aroca, Rafael Vidal and Luiz Marcos Garcia Gonçalves (2012). "Towards green data centers: A comparison of x86 and ARM architectures power efficiency". In: *Journal of Parallel and Distributed Computing* 72.12, pp. 1770–1780. ISSN: 0743-7315. DOI: https://doi.org/10.1016/j.jpdc.2012.08.005. URL: https://www.sciencedirect.com/science/article/pii/S0743731512002122.

Aversa, Rocco et al. (Jan. 2004). "High Performance Computing: Paradigm and Infrastructure". In: ISBN: 047165471X.

Ayala, Alan et al. (2022). *Analysis of the Communication and Computation Cost of FFT Libraries towards Exascale*. Tech. rep. Technical Report ICL-UT-22-07. https://icl. utk. edu/files/publications/2022 ...

Bertolli, Carlo et al. (2014). "Coordinating GPU Threads for OpenMP 4.0 in LLVM". In: *2014 LLVM Compiler Infrastructure in HPC*, pp. 12–21. DOI: 10.1109/LLVM-HPC.2014.10.

Betkaoui, Brahim, David B Thomas, and Wayne Luk (2010). "Comparing performance and energy efficiency of FPGAs and GPUs for high productivity computing". In: *2010 International Conference on Field-Programmable Technology*. IEEE, pp. 94–101.

Bolz, Jeff et al. (July 2003). "Sparse matrix solvers on the GPU: conjugate gradients and multigrid". In: *ACM Trans. Graph.* 22.3, pp. 917–924. ISSN: 0730-0301. DOI: 10.1145/882262.882364. URL: https://doi.org/10.1145/882262.882364.

Born, Max and Emil Wolf (2013). *Principles of optics: electromagnetic theory of propagation, interference and diffraction of light*. Elsevier.

Briggs, Daniel Shenon (Jan. 1995). "High fidelity deconvolution of moderately resolved sources". PhD thesis. New Mexico Institute of Mining and Technology.

Brunet, T. et al. (2024). "Quantum radio astronomy: Data encodings and quantum image processing". In: *Astronomy and Computing* 47, p. 100796. ISSN: 2213-1337. DOI: https://doi.org/10.1016/j.ascom.2024.100796. URL: https://www.sciencedirect.com/science/article/pii/S2213133724000118.

Budruk, Ravi, Don Anderson, and Tom Shanley (2004). *PCI express system architecture*. Addison-Wesley Professional.

Calore, Enrico et al. (2020). "ThunderX2 Performance and Energy-Efficiency for HPC Workloads". In: *Computation* 8.1. ISSN: 2079-3197. DOI: 10.3390/computation8010020. URL: https://www.mdpi.com/2079-3197/8/1/20.

Cambier, Leopold, Doris Pan, and Lukasz Ligowski (2022). "Multinode multi-gpu: Using nvidia cufftmp ffts at scale". In: *NVIDIA Technical Blog*.

Chandra, Robit et al. (2001). *Parallel programming in OpenMP*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc. ISBN: 1558606718.

Chandrasekaran, Sunita and Guido Juckeland (2017). *OpenACC for programmers: concepts and strategies*. Addison-Wesley Professional.

Chapman, Barbara, Gabriele Jost, and Ruud van der Pas (2007). *Using OpenMP: Portable Shared Memory Parallel Programming (Scientific and Engineering Computation)*. The MIT Press. ISBN: 0262533022.

Chen, Chen-Chun et al. (2023). "Implementing and Optimizing a GPU-aware MPI Library for Intel GPUs: Early Experiences". In: *2023 IEEE/ACM 23rd International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*. IEEE, pp. 131–140.

Clarke, Lyndon, Ian Glendinning, and Rolf Hempel (1994). "The MPI message passing interface standard". In: *Programming Environments for Massively Parallel Distributed Systems: Working Conference of the IFIP WG 10.3, April 25–29, 1994*. Springer, pp. 213–218.

Cornwell, T. (n.d.). "Deconvolution". In: pp. 151–170.

Cornwell, T. J., K. Golap, and S. Bhatnagar (Nov. 2008). "The Noncoplanar Baselines Effect in Radio Interferometry: The W-Projection Algorithm". In: *IEEE Journal of Selected Topics in Signal Processing* 2.5, pp. 647–657. DOI: 10.1109/JSTSP.2008.2005290. arXiv: 0807.4161 [astro-ph].

Cornwell, TJ and RA Perley (1992). "Radio-interferometric imaging of very large fields-The problem of non-coplanar arrays". In: *Astronomy and Astrophysics (ISSN 0004-6361), vol. 261, no. 1, p. 353-364.* 261, pp. 353–364.

Cornwell, TJ and PN Wilkinson (1981). "A new method for making maps with unstable radio interferometers". In: *Monthly Notices of the Royal Astronomical Society* 196.4, pp. 1067–1086.

Dakić, Vedran et al. (2024). "Evaluating ARM and RISC-V Architectures for High-Performance Computing with Docker and Kubernetes". In: *Electronics* 13.17. ISSN: 2079-9292. DOI: 10.3390/electronics13173494. URL: https://www.mdpi.com/2079-9292/13/17/3494.

Das, Monika (n.d.). "Performance Evaluation of Fast Fourier Transform (FFT) Libraries for High Performance Computing". PhD thesis. Université de Bourgogne.

De Rubeis, Emanuele et al. (2024). "Accelerating Radio Astronomy Imaging with Rick". In: *Available at SSRN*. DOI: 10.2139/ssrn.4933256. URL: http://dx.doi.org/10.2139/ssrn.4933256.

Dennard, R.H. et al. (1974). "Design of ion-implanted MOSFET's with very small physical dimensions". In: *IEEE Journal of Solid-State Circuits* 9.5, pp. 256–268. DOI: 10.1109/JSSC.1974.1050511.

Dennard, R.H. et al. (1999). "Design Of Ion-implanted MOSFET's with Very Small Physical Dimensions". In: *Proceedings of the IEEE* 87.4, pp. 668–678. DOI: 10.1109/JPROC.1999.752522.

Design, BIOS (1997). "AMD". In.

Desrochers, Spencer, Chad Paradis, and Vincent M Weaver (2016). "A validation of DRAM RAPL power measurements". In: *Proceedings of the Second International Symposium on Memory Systems*, pp. 455–470.

Doerfert, Johannes et al. (2022). "Co-Designing an OpenMP GPU runtime and optimizations for near-zero overhead execution". In: *2022 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, pp. 504–514.

Dongarra, Jack, Piotr Luszczek, and M Heroux (2013). "HPCG technical specification". In: *Sandia National Laboratories, Sandia Report SAND2013-8752*.

Dongarra, Jack J. (1988). "The LINPACK Benchmark: An explanation". In: *Supercomputing*. Ed. by E. N. Houstis, T. S. Papatheodorou, and C. D. Polychronopoulos. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 456–474. ISBN: 978-3-540-38888-3.

Dongarra, Jack J., Piotr Luszczek, and Antoine Petitet (2003). "The LINPACK Benchmark: past, present and future". In: *Concurrency and Computation: Practice and Experience* 15.9, pp. 803–820. DOI: https://doi.org/10.1002/cpe.728. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/cpe.728. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/cpe.728.

Eijkhout, Victor, Robert van de Geijn, and Edmond Chow (Jan. 2016). *Introduction to High Performance Scientific Computing*. DOI: 10.5281/zenodo.49897.

Emonts, B. et al. (2019). *The CASA software for radio astronomy: status update from ADASS 2019*. arXiv: 1912.09437 [astro-ph.IM]. URL: https://arxiv.org/abs/1912.09437.

Esmaeilzadeh, Hadi et al. (June 2011). "Dark silicon and the end of multicore scaling". In: *SIGARCH Comput. Archit. News* 39.3, pp. 365–376. ISSN: 0163-5964. DOI: 10.1145/2024723.2000108. URL: https://doi.org/10.1145/2024723.2000108.

Faraji, Iman and Ahmad Afsahi (2014). "GPU-aware intranode MPI_Allreduce". In: *Proceedings of the 21st European MPI Users' Group Meeting*, pp. 45–50.

Faraji, Iman and Ahmad Afsahi (2018). "Design considerations for GPU-aware collective communications in MPI". In: *Concurrency and Computation: Practice and Experience* 30.17, e4667.

Farber, Rob (2016). *Parallel programming with OpenACC*. Newnes.

Feng, Wu-chun and Kirk Cameron (2007). "The green500 list: Encouraging sustainable supercomputing". In: *Computer* 40.12, pp. 50–55.

Feng, Wu-chun and Heshan Lin (2010). "The green500 list: Year two". In: *2010 IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum (IPDPSW)*. IEEE, pp. 1–8.

Feng, Wu-Chun and Thomas Scogland (2009). "The green500 list: Year one". In: *2009 IEEE International Symposium on Parallel & Distributed Processing*. IEEE, pp. 1–7.

Fountain, Trevor, Alexandra McCarthy, Fangfang Peng, et al. (2005). "PCI express: An overview of PCI express, cabled PCI express and PXI express". In: *10th ICALEPCS Int. Conf. on Accelerator & Large Expt. Physics Control Systems*.

Gai, Keke et al. (2016). "Dynamic energy-aware cloudlet-based mobile cloud computing model for green computing". In: *Journal of Network and Computer Applications* 59, pp. 46–54. ISSN: 1084-8045. DOI: https://doi.org/10.1016/j.jnca.2015.05.016. URL: https://www.sciencedirect.com/science/article/pii/S108480451500123X.

Ge, R et al. (2007). "Power measurement tutorial for the Green500 list". In: *The Green500 List: Environmentally Responsible Supercomputing*.

Geveler, Markus et al. (2017). "The icarus white paper: a scalable, energy-efficient, solar-powered hpc center based on low power gpus". In: *Euro-Par 2016: Parallel Processing Workshops: Euro-Par 2016 International Workshops, Grenoble, France, August 24-26, 2016, Revised Selected Papers 22*. Springer, pp. 737–749.

Gheller, Claudio, Giuliano Taffoni, and David Goz (Jan. 2023). "High performance w-stacking for imaging radio astronomy data: a parallel and accelerated solution". In: *RAS Techniques and Instruments* 2.1, pp. 91–105. DOI: 10.1093/rasti/rzad002. arXiv: 2301.06061 [astro-ph.IM].

Gheller, Claudio et al. (2024). "HPC and GPU Accelerated Imaging Toward the SKA Era". In: *Proceedings Volume Software and Cyberinfrastructure for Astronomy VIII*. Vol. 13101, 131011H. DOI: 10.1117/12.3019055. URL: https://doi.org/10.1117/12.3019055.

Godoy, William F. et al. (2020). "ADIOS 2: The Adaptable Input Output System. A framework for high-performance data management". In: *SoftwareX* 12, p. 100561. ISSN: 2352-7110. DOI: https://doi.org/10.1016/j.softx.2020.100561. URL: https://www.sciencedirect.com/science/article/pii/S2352711019302560.

Gong, Qian et al. (2023). "MGARD: A multigrid framework for high-performance, error-controlled data compression and refactoring". In: *SoftwareX* 24, p. 101590.

Gustafson, John L. (May 1988). "Reevaluating Amdahl's law". In: *Commun. ACM* 31.5, pp. 532–533. ISSN: 0001-0782. DOI: 10.1145/42411.42415. URL: https://doi.org/10.1145/42411.42415.

Hähnel, Marcus et al. (2012). "Measuring energy consumption for short code paths using RAPL". In: *ACM SIGMETRICS Performance Evaluation Review* 40.3, pp. 13–17.

Hanford, Nathan et al. (2020). "Challenges of gpu-aware communication in mpi". In: *2020 Workshop on Exascale MPI (ExaMPI)*. IEEE, pp. 1–10.

Hardy, Stephen J (2013). "Direct deconvolution of radio synthesis images using l1 minimisation". In: *Astronomy & Astrophysics* 557, A134.

Hayashi, Akihiro et al. (Jan. 2019). "Performance evaluation of OpenMP's target construct on GPUs - exploring compiler optimisations". In: *International Journal of High Performance Computing and Networking* 13, p. 54. DOI: 10.1504/IJHPCN.2019.097051.

He, Shuaiming, Wei Wan, and Junhong Li (2024). "Network communication optimization of RCCL communication library in Multi-NIC systems". In: *Third International Conference on Algorithms, Microchips, and Network Applications (AMNA 2024)*. Vol. 13171. SPIE, pp. 418–425.

Heideman, M., D. Johnson, and C. Burrus (1984). "Gauss and the history of the fast fourier transform". In: *IEEE ASSP Magazine* 1.4, pp. 14–21. DOI: 10.1109/MASSP.1984.1162257.

Henning, J.L. (2000). "SPEC CPU2000: measuring CPU performance in the New Millennium". In: *Computer* 33.7, pp. 28–35. DOI: 10.1109/2.869367.

Herdman, J. A. et al. (2014). "Achieving Portability and Performance through OpenACC". In: *2014 First Workshop on Accelerator Programming using Directives*, pp. 19–26. DOI: 10.1109/WACCPD.2014.10.

Heroux, Michael Allen, Jack Dongarra, and Piotr Luszczek (2013). *HPCG benchmark technical specification*. Tech. rep. Sandia National Lab.(SNL-NM), Albuquerque, NM (United States).

Högbom, JA (1974). "Aperture synthesis with a non-regular distribution of interferometer baselines". In: *Astronomy and Astrophysics Supplement, Vol. 15, p. 417* 15, p. 417.

Hsu, Chung-Hsing and Neena Imam (2021). "Assessment of nvshmem for high performance computing". In: *International Journal of Networking and Computing* 11.1, pp. 78–101.

Hsu, Chung-Hsing et al. (2020). "An initial assessment of NVSHMEM for high performance computing". In: *2020 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE, pp. 1–10.

Huang, Song, Shucai Xiao, and Wu-chun Feng (2009). "On the energy efficiency of graphics processing units for scientific computing". In: *2009 IEEE International Symposium on Parallel & Distributed Processing*. IEEE, pp. 1–8.

Huber, Joseph et al. (2022). "Efficient execution of OpenMP on GPUs". In: *Proceedings of the 20th IEEE/ACM International Symposium on Code Generation and Optimization*. CGO '22. Virtual Event, Republic of Korea: IEEE Press, pp. 41–52. ISBN: 9781665405843. DOI: 10.1109/CGO53902.2022.9741290. URL: https://doi.org/10.1109/CGO53902.2022.9741290.

Iannello, Giulio (1997). "Efficient algorithms for the reduce-scatter operation in LogGP". In: *IEEE Transactions on Parallel and Distributed Systems* 8.9, pp. 970–982.

Jackson, J. et al. (Aug. 1991). "On dense granular flows". In: *Medical Imaging, IEEE Trans. on*, 10, p. 473.

Jaeger, S. (Aug. 2008). "The Common Astronomy Software Application (CASA)". In: *Astronomical Data Analysis Software and Systems XVII*. Ed. by R. W. Argyle, P. S. Bunclark, and J. R. Lewis. Vol. 394. Astronomical Society of the Pacific Conference Series, p. 623.

Jahanshahi, Ali et al. (2020). "Gpu-nest: Characterizing energy efficiency of multi-gpu inference servers". In: *IEEE Computer Architecture Letters* 19.2, pp. 139–142.

Johnston, S. et al. (Dec. 2007). "Science with the Australian Square Kilometre Array Pathfinder". In: 24.4, pp. 174–188. DOI: 10.1071/AS07033. arXiv: 0711.2103 [astro-ph].

Jonas, J. and MeerKAT Team (Jan. 2016). "The MeerKAT Radio Telescope". In: *MeerKAT Science: On the Pathway to the SKA*, 1, p. 1.

Kasichayanula, Kiran et al. (2012). "Power aware computing on GPUs". In: *2012 Symposium on Application Accelerators in High Performance Computing*. IEEE, pp. 64–73.

Keckler, Stephen et al. (Nov. 2011). "GPUs and the Future of Parallel Computing". In: *Micro, IEEE* 31, pp. 7–17. DOI: 10.1109/MM.2011.89.

Kogler, Andreas et al. (2022). "Finding and Exploiting CPU Features using MSR Templating". In: *2022 IEEE Symposium on Security and Privacy (SP)*. IEEE, pp. 1474–1490.

Krüger, Jens and Rüdiger Westermann (July 2003). "Linear algebra operators for GPU implementation of numerical algorithms". In: *ACM Trans. Graph.* 22.3, pp. 908–916. ISSN: 0730-0301. DOI: 10.1145/882262.882363. URL: https://doi.org/10.1145/882262.882363.

Kurp, Patrick (2008). "Green computing". In: *Communications of the ACM* 51.10, pp. 11–13.

Laufer, Michael and Erick Fredj (2022). "High performance parallel I/O and in-situ analysis in the WRF model with ADIOS2". In: *arXiv preprint arXiv:2201.08228*.

Lee, Seyong and Rudolf Eigenmann (2010). "OpenMPC: Extended OpenMP programming and tuning for GPUs". In: *SC'10: Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, pp. 1–11.

Lee, Seyong, Jungwon Kim, and Jeffrey S. Vetter (2016). "OpenACC to FPGA: A Framework for Directive-Based High-Performance Reconfigurable Computing". In: *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pp. 544–554. DOI: 10.1109/IPDPS.2016.28.

Lee, Seyong, Seung-Jai Min, and Rudolf Eigenmann (2009). "OpenMP to GPGPU: a compiler framework for automatic translation and optimization". In: *ACM Sigplan Notices* 44.4, pp. 101–110.

Leland, Robert et al. (Jan. 2014). "Large-Scale Data Analytics and Its Relationship to Simulation". In.

Li, Ang et al. (2019). "Evaluating Modern GPU Interconnect: PCIe, NVLink, NV-SLI, NVSwitch and GPUDirect". In: *IEEE Transactions on Parallel and Distributed Systems* 31, pp. 94–110. URL: https://api.semanticscholar.org/CorpusID:75136160.

Li, Feng, Tim J Cornwell, and Frank de Hoog (2011). "The application of compressive sampling to radio astronomy-i. deconvolution". In: *Astronomy & Astrophysics* 528, A31.

Liu, Jiuxing et al. (2003). "High performance RDMA-based MPI implementation over InfiniBand". In: *Proceedings of the 17th annual international conference on Supercomputing*, pp. 295–304.

Lo, Chia-Tien Dan and Kai Qian (2010). "Green computing methodology for next generation computing scientists". In: *2010 IEEE 34th Annual Computer Software and Applications Conference*. IEEE, pp. 250–251.

Manavski, Svetlin and Giorgio Valle (Feb. 2008). "CUDA compatible GPU cards as efficient hardware accelerators for Smith-Waterman string alignment". In: *BMC bioinformatics* 9 Suppl 2, S10. DOI: 10.1186/1471-2105-9-S2-S10.

Marjanović, Vladimir, José Gracia, and Colin W Glass (2015). "Performance modeling of the HPCG benchmark". In: *High Performance Computing Systems. Performance Modeling, Benchmarking, and Simulation: 5th International Workshop, PMBS 2014, New Orleans, LA, USA, November 16, 2014. Revised Selected Papers 5*. Springer, pp. 172–192.

Martinelli, Alberto Riccardo et al. (2023). "CAPIO: a middleware for transparent I/O streaming in data-intensive workflows". In: *2023 IEEE 30th International Conference on High Performance Computing, Data, and Analytics (HiPC)*. IEEE, pp. 153–163.

McCraw, Heike et al. (2014). "Power monitoring with PAPI for extreme scale architectures and dataflow-based programming models". In: *2014 IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE, pp. 385–391.

Mickaelian, Areg M (2016). "Astronomical surveys and big data". In: *Open Astronomy* 25.1, pp. 75–88.

Mitchell, D. et al. (May 2010). "The Murchison Widefield Array". In: *RFI Mitigation Workshop*, 16, p. 16. arXiv: 1008.2551 [astro-ph.IM].

Mittal, Sparsh and Jeffrey S Vetter (2014). "A survey of methods for analyzing and improving GPU energy efficiency". In: *ACM Computing Surveys (CSUR)* 47.2, pp. 1–23.

Monaco, Pierluigi, Tom Theuns, and Giuliano Taffoni (2002). "The pinocchio algorithm: pinpointing orbit-crossing collapsed hierarchical objects in a linear density field". In: *Monthly Notices of the Royal Astronomical Society* 331.3, pp. 587–608.

Moore, G.E. (1998). "Cramming More Components Onto Integrated Circuits". In: *Proceedings of the IEEE* 86.1, pp. 82–85. DOI: 10.1109/JPROC.1998.658762.

Moore, Gordon E. (2006). "Cramming more components onto integrated circuits, Reprinted from Electronics, volume 38, number 8, April 19, 1965, pp.114 ff." In: *IEEE Solid-State Circuits Society Newsletter* 11.3, pp. 33–35. DOI: 10.1109/N-SSC.2006.4785860.

More, Nitin S and Rajesh B Ingle (2017). "Challenges in green computing for energy saving techniques". In: *2017 International Conference on Emerging Trends & Innovation in ICT (ICEI)*. IEEE, pp. 73–76.

Mucci, Philip J et al. (1999). "PAPI: A portable interface to hardware performance counters". In: *Proceedings of the department of defense HPCMP users group conference*. Vol. 710.

Nguyen, Tan et al. (2020). "The performance and energy efficiency potential of FPGAs in scientific computing". In: *2020 IEEE/ACM Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS)*. IEEE, pp. 8–19.

Nguyen, Tan et al. (2022). "FPGA-based HPC accelerators: An evaluation on performance and energy efficiency". In: *Concurrency and Computation: Practice and Experience* 34.20, e6570.

Nielsen, Frank (Feb. 2016). "Introduction to MPI: The Message Passing Interface". In: pp. 21–62. ISBN: 978-3-319-21902-8. DOI: 10.1007/978-3-319-21903-5_2.

Noor Rahman, Tahmid, Nusaiba Khan, and Zarif Zaman (Jan. 2024). "Redefining Computing: Rise of ARM from consumer to Cloud for energy efficiency". In: *World Journal of Advanced Research and Reviews* 21, pp. 817–835. DOI: 10.30574/wjarr.2024.21.1.0017.

Nussbaumer, Henri J and Henri J Nussbaumer (1982). *The fast Fourier transform*. Springer.

NVIDIA, NCCL (2017). *NVIDIA Collective Communications Library (NCCL)*.

Nvidia, NCCL (n.d.). *Optimized primitives for collective multi-GPU communication*.

Offringa, A. R., B. McKinley, Hurley-Walker, et al. (2014). "WSClean: an implementation of a fast, generic wide-field imager for radio astronomy". In: *MNRAS* 444.1, pp. 606–619. DOI: 10.1093/mnras/stu1368.

Offringa, A. R. and O. Smirnov (2017). "An optimized algorithm for multiscale wideband deconvolution of radio astronomical images". In: *MNRAS* 471.1, pp. 301–316. DOI: 10.1093/mnras/stx1547.

Offringa, AR and O Smirnov (2017). "An optimized algorithm for multiscale wideband deconvolution of radio astronomical images". In: *Monthly Notices of the Royal Astronomical Society* 471.1, pp. 301–316.

Owens, John D et al. (2007). "A survey of general-purpose computation on graphics hardware". In: *Computer graphics forum*. Vol. 26. 1. Wiley Online Library, pp. 80–113.

Owens, John D et al. (2008). "GPU computing". In: *Proceedings of the IEEE* 96.5, pp. 879–899.

Oyarzun, Guillermo, Daniel Mira, and Guillaume Houzeaux (2021). "Performance assessment of CUDA and OpenACC in large scale combustion simulations". In: *CoRR* abs/2107.11541. arXiv: 2107.11541. URL: https://arxiv.org/abs/2107.11541.

Parashar, Varun and Vivek Kumar (2023). "Improving energy efficiency of nvidia GPUs". In.

Pas, Ruud van der (2002). In: *Memory Hierarchy in Cache-Based Systems*. URL: https://api.semanticscholar.org/CorpusID:18765250.

Pas, Ruud van der, Eric Stotzer, and Christian Terboven (Oct. 2017). *Using OpenMP—The Next Step: Affinity, Accelerators, Tasking, and SIMD*. The MIT Press. ISBN: 9780262344012. DOI: 10.7551/mitpress/10031.001.0001. URL: https://doi.org/10.7551/mitpress/10031.001.0001.

Paul, Showmick Guha et al. (2023). "A Comprehensive Review of Green Computing: Past, Present, and Future Research". In: *IEEE Access* 11, pp. 87445–87494. DOI: 10.1109/ACCESS.2023.3304332.

Poeschel, Franz et al. (2021). "Transitioning from file-based HPC workflows to streaming data pipelines with openPMD and ADIOS2". In: *Smoky Mountains Computational Sciences and Engineering Conference*. Springer, pp. 99–118.

Potluri, Sreeram et al. (2015). "Exploring OpenSHMEM model to program GPU-based extreme-scale systems". In: *OpenSHMEM and Related Technologies. Experiences, Implementations, and Technologies: Second Workshop, OpenSHMEM 2015, Annapolis, MD, USA, August 4-6, 2015. Revised Selected Papers 1*. Springer, pp. 18–35.

Qasaimeh, Murad et al. (2019). "Comparing energy efficiency of CPU, GPU and FPGA implementations for vision kernels". In: *2019 IEEE international conference on embedded software and systems (ICESS)*. IEEE, pp. 1–8.

Raffin, Guillaume and Mathilde Jay (2023). "NVML sensor (CPU+ GPU)". In.

Saavedra, R.H. and A.J. Smith (1995). "Measuring cache and TLB performance and their effect on benchmark runtimes". In: *IEEE Transactions on Computers* 44.10, pp. 1223–1235. DOI: 10.1109/12.467697.

Sabne, Amit et al. (2015). "Evaluating performance portability of OpenACC". In: *Languages and Compilers for Parallel Computing: 27th International Workshop, LCPC 2014, Hillsboro, OR, USA, September 15-17, 2014, Revised Selected Papers 27*. Springer, pp. 51–66.

Sanders, J. and E. Kandrot (2010). *CUDA by Example: An Introduction to General-Purpose GPU Programming*. Pearson Education. ISBN: 9780132180139. URL: https://books.google.it/books?id=49OmnOmTEtQC.

Sault, Robert John and TA Oosterloo (2007). "Imaging algorithms in radio interferometry". In: *arXiv preprint astro-ph/0701171*.

Schöne, Robert et al. (2021). "Energy Efficiency Aspects of the AMD Zen 2 Architecture". In: *CoRR* abs/2108.00808. arXiv: 2108.00808. URL: https://arxiv.org/abs/2108.00808.

Scogland, Tom, Balaji Subramaniam, and Wu-chun Feng (2011). "Emerging trends on the evolving green500: Year three". In: *2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum*. IEEE, pp. 822–828.

Scogland, Tom, Balaji Subramaniam, and Wu-chun Feng (2013). "The Green500 list: escapades to exascale". In: *Computer Science-Research and Development* 28, pp. 109–117.

Segal, Mark and Kurt Akeley (2004). "The OpenGL Graphics System: A Specification". In: URL: https://api.semanticscholar.org/CorpusID:61070754.

Sensi, Daniele De et al. (2024). *Exploring GPU-to-GPU Communication: Insights into Supercomputer Interconnects*. arXiv: 2408.14090 [cs.DC]. URL: https://arxiv.org/abs/2408.14090.

Shafie Khorassani, Kawthar et al. (2021). "Designing a ROCm-aware MPI library for AMD GPUs: early experiences". In: *International Conference on High Performance Computing*. Springer, pp. 118–136.

Shan, Hongzhang, Samuel Williams, and Calvin W. Johnson (2018). "Improving MPI Reduction Performance for Manycore Architectures with OpenMP and Data Compression". In: *2018 IEEE/ACM Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS)*, pp. 1–11. DOI: 10.1109/PMBS.2018.8641632.

Springel, Volker (2005). "The cosmological simulation code GADGET-2". In: *Monthly notices of the royal astronomical society* 364.4, pp. 1105–1134.

Starck, Jean-Luc, Eric Pantin, and Fionn Murtagh (2002). "Deconvolution in astronomy: A review". In: *Publications of the Astronomical Society of the Pacific* 114.800, p. 1051.

Suárez, Daniel, Francisco Almeida, and Vicente Blanco (Feb. 2024). "Comprehensive analysis of energy efficiency and performance of ARM and RISC-V SoCs". In: *The Journal of Supercomputing* 80, pp. 1–19. DOI: 10.1007/s11227-024-05946-9.

Sun, Yifan, Trinayan Baruah, and David Kaeli (Dec. 2022). *Accelerated Computing with HIP*. ISBN: 979-8218107444.

Sur, Sayantan, Matthew J. Koop, and Dhabaleswar K. Panda (2006). "High-performance and scalable MPI over InfiniBand with reduced memory usage: an in-depth performance analysis". In: *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*. SC '06. Tampa, Florida: Association for Computing Machinery, 105–es. ISBN: 0769527000. DOI: 10.1145/1188455.1188565. URL: https://doi.org/10.1145/1188455.1188565.

Tarditi, David, Sidd Puri, and Jose Oglesby (Oct. 2006). "Accelerator: using data parallelism to program GPUs for general-purpose uses". In: *SIGARCH Comput. Archit. News* 34.5, pp. 325–335. ISSN: 0163-5964. DOI: 10.1145/1168919.1168898. URL: https://doi.org/10.1145/1168919.1168898.

Team, The CASA et al. (Nov. 2022). "CASA, the Common Astronomy Software Applications for Radio Astronomy". In: *Publications of the Astronomical Society of the Pacific* 134.1041, p. 114501. ISSN: 1538-3873. DOI: 10.1088/1538-3873/ac9642. URL: http://dx.doi.org/10.1088/1538-3873/ac9642.

Terpstra, Dan et al. (2010). "Collecting Performance Data with PAPI-C". In: *Tools for High Performance Computing 2009*. Ed. by Matthias S. Müller et al. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 157–173. ISBN: 978-3-642-11261-4.

The MPI Forum, CORPORATE (1993). "MPI: a message passing interface". In: *Proceedings of the 1993 ACM/IEEE Conference on Supercomputing*. Supercomputing '93. Portland, Oregon, USA: Association for Computing Machinery, pp. 878–883. ISBN: 0818643404. DOI: 10.1145/169627.169855. URL: https://doi.org/10.1145/169627.169855.

Van der Tol, Sebastiaan, Bram Veenboer, and André R Offringa (2018a). "Image Domain Gridding: a fast method for convolutional resampling of visibilities". In: *Astronomy & Astrophysics* 616, A27.

Van der Tol, Sebastiaan, Bram Veenboer, and André R. Offringa (2018b). "Image Domain Gridding: a fast method for convolutional resampling of visibilities". In: *A&A* 616, A27. DOI: 10.1051/0004-6361/201832858. URL: https://doi.org/10.1051/0004-6361/201832858.

van Haarlem, M. P. et al. (Aug. 2013). "LOFAR: The LOw-Frequency ARray". In: 556, A2, A2. DOI: 10.1051/0004-6361/201220873. arXiv: 1305.3550 [astro-ph.IM].

Verma, Manthan et al. (2023). "Scalable Multi-node Fast Fourier Transform on GPUs". In: *SN Computer Science* 4.5, p. 625.

Walker, David W. (1992). "Standards for message-passing in a distributed memory environment". In: URL: https://api.semanticscholar.org/CorpusID:62765194.

Wang, Bin et al. (2013). "Exploring hybrid memory for GPU energy efficiency through software-hardware co-design". In: *Proceedings of the 22nd international conference on Parallel architectures and compilation techniques*. IEEE, pp. 93–102.

Wang, Hao et al. (2013). "GPU-aware MPI on RDMA-enabled clusters: Design, implementation and evaluation". In: *IEEE Transactions on Parallel and Distributed Systems* 25.10, pp. 2595–2605.

Weaver, Vincent M et al. (2012). "Measuring energy and power with PAPI". In: *2012 41st international conference on parallel processing workshops*. IEEE, pp. 262–268.

Wienke, Sandra et al. (2012). "OpenACC — First Experiences with Real-World Applications". In: *Euro-Par 2012 Parallel Processing*. Ed. by Christos Kaklamanis, Theodore Papatheodorou, and Paul G. Spirakis. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 859–870. ISBN: 978-3-642-32820-6.

Wilen, Adam, Justin Schade, and Ron Thornburg (2003). *Introduction to PCI Express*. Intel Press Santa Clara.

Williamson, Alexander et al. (2024). *Optimising the Processing and Storage of Radio Astronomy Data*. arXiv: 2410.02285 [astro-ph.IM].

Wrinkler, Frank (2020). "Redesigning PAPI's high-level API". In: *University of Tennessee, Tech. Rep. ICL-UT-20–03*.

Yatawatta, Sarod (2014). "Adaptive weighting in radio interferometric imaging". In: *Monthly Notices of the Royal Astronomical Society* 444.1, pp. 790–796.

Ye, Haoyang et al. (2020). "Optimal gridding and degridding in radio interferometry imaging". In: *Monthly Notices of the Royal Astronomical Society* 491.1, pp. 1146–1159.

Zajqc, Piotr, Melvin Galicia, and Andrzej Napieralski (2018). "Thermal-aware Floorplanning Guidelines for 3D ICs with Integrated Microchannels". In: *2018 25th International Conference "Mixed Design of Integrated Circuits and System" (MIXDES)*, pp. 258–261. DOI: 10.23919/MIXDES.2018.8436886.