RAS Techniques and Instruments

RASTAI **2,** 91–105 (2023) https://doi.org/10.1093/rasti/rzad002
Advance Access publication 2023 February 2

High performance *w*-stacking for imaging radio astronomy data: a parallel and accelerated solution

Claudio Gheller [1]★ Giuliano Taffoni[2] and David Goz[2]

[1] *Institute of Radioastronomy, INAF, Via Gobetti 101, I-40121 Bologna, Italy*
[2] *Astronomical Observatory of Trieste INAF, Via GB Tiepolo 11, I-34143 Trieste, Italy*

Accepted 2023 January 9. Received 2022 December 3; in original form 2022 February 27

ABSTRACT

Current and upcoming radio-interferometers are expected to produce volumes of data of increasing size that need to be processed in order to generate the corresponding sky brightness distributions through imaging. This represents an outstanding computational challenge, especially when large fields of view and/or high-resolution observations are processed. We have investigated the adoption of modern high performance computing systems specifically addressing the gridding, fast Fourier transform, and *w*-correction of imaging, combining parallel and accelerated solutions. We have demonstrated that the code we have developed can support data set and images of any size compatible with the available hardware, efficiently scaling up to thousands of cores or hundreds of graphic processing units, keeping the time to solution <1 h even when images of the size of the order of billions or tens of billions of pixels are generated. In addition, portability has been targeted as a primary objective, both in terms of usability on different computing platforms and in terms of performance. The presented results have been obtained on two different state-of-the-art high performance computing architectures.

Key words: software: data analysis – software: development – techniques: image processing – techniques: interferometric.

1. INTRODUCTION

In the next decade, current and upcoming radio-interferometers, like the Low Frequency Array (LOFAR; van Haarlem et al. 2013), MeerKAT (Jonas & MeerKAT Team 2016), the Murchison Widefield Array (Mitchell et al. 2010), and the Australian Square Kilometre Array Pathfinder (ASKAP; Johnston et al. 2007), in the perspective of the Square Kilometre Array (SKA MID and LOW),[1] will produce huge volumes of data. This will represent not only an invaluable opportunity for scientists, but also an outstanding technological challenge. The expected data volume will be hard to manage with traditional approaches. Data will have to be stored in dedicated facilities, providing the necessary capacity at the highest performance. Corresponding data processing will have to be performed local to the data, exploiting available high performance computing (HPC) resources. Data reduction and imaging software tools will have to be adapted, if not completely re-designed, in order to efficiently run at scale.

In modern HPC systems, performance is being achieved through many-core and accelerated computing [based, for instance, on graphic processing units (GPUs)], and current trends suggest that some form of heterogeneous computing will continue to be prevalent in emerging architectures. Therefore, the ability to fully exploit new heterogeneous and many-core solutions is of paramount importance towards achieving optimal performance. On the other hand, with the increasing size and complexity of observational data, it is of primary importance for scientists to be able to exploit all available hardware in emerging HPC environments. Exploiting these novel hybrid architectures is non-trivial however, due to the challenges presented by mixed hardware computing and the increasing levels of architectural parallelism. New algorithms and numerical and computational solutions are required.

In radio interferometry, data processing is generally performed in various steps (often repeated iteratively), based on the work of Hamaker, Bregman & Sault (1996), Rau et al. (2009), and Smirnov (2011a, b). Firstly, the calibration step aims at estimating and correcting for time-, frequency-, antenna-, and direction-dependent instrumental errors (Bhatnagar et al. 2008; Cornwell, Golap & Bhatnagar 2008; Bhatnagar, Rau & Golap 2013; Tasse et al. 2013; Bhatnagar & Cornwell 2017; Jagannathan et al. 2017). Various software tools address calibration, like DPPP(van Diepen, Dijkema & Offringa 2018), CUBICAL (Kenyon et al. 2018), or KILLMS.[2] This is followed by *imaging*, i.e. the processes of Fourier-transforming the calibrated visibilities into images [e.g. DDFACET, addressing anyway also calibration (Tasse et al. 2018), and WSCLEAN (Offringa et al. 2014)]. Then deconvolution (see Cornwell 1999) corrects the resulting images for the incomplete sampling of the Fourier plane. Deconvolution is implemented in terms of the consolidated and widely used Clean algorithm (Högbom 1974) in its different variants (Clark 1980; Schwab 1984, among the most widely used), or other approaches, like the maximum entropy model (Cornwell & Evans

★ E-mail: claudio.gheller@gmail.com
[1] https://www.skatelescope.org/
[2] https://github.com/saopicc/killMS

1985), the MORESANE model (Dabbech et al. 2015), and the sparsity averaging reweighted analysis family of deconvolution algorithms defined in the context of optimization theory, and implemented in the PURI-PSI library,[3] have been demonstrated to bring joint precision and scalability of the reconstruction process (Carrillo, McEwen & Wiaux 2012, 2014; Onose et al. 2016; Pratley et al. 2018; Thouvenin et al. 2020, 2022), also including a joint approach for calibration and deconvolution (Repetti et al. 2017; Dabbech et al. 2021), the SASIR method (Girard et al. 2015), the RESOLVE Bayesian method (Junklewitz et al. 2016), and the Clean multiscale deconvolution approach (Hoekstra et al. 2005; Rau & Cornwell 2011). Finally, denoising (we refer to Roscani et al. 2020, for a comprehensive review) and source detection and characterization are performed using tools that are usually available within source finding software packages, like PYBDSF (Mostert et al. 2021), SOFIA (Serra et al. 2015), and AEGEAN (Hancock et al. 2012) among the most up-to-date. Also of note are comprehensive and widely adopted software platforms, able to perform most of the previous tasks, like CASA (McMullin et al. 2007), AIPS (Greisen 2003), MIRIAD (Sault, Teuben & Wright 1995), and ASKAPSOFT (Wieringa, Raja & Ord 2020). Finally, it is worth mentioning the two first applications of deep learning for imaging presented by Terris et al. (2022) and Dabbech et al. (2022).

Imaging represents one of the most computational demanding steps of the processing pipeline, both in terms of memory request and in terms of computing time, associated to operations like gridding, i.e. the convolutional resampling of the observed data on a computational mesh, or fast Fourier transform (FFT) transforming, to move from Fourier to real space and vice-versa. The computational requirements increase further when observations with large fields of view (FoVs; as typically happens with current radio-interferometers and at the lowest frequencies) are considered, since curvature effects cannot be neglected and the problem becomes fully three-dimensional (3D), with the introduction of the w-term correction (see Section 2, Cornwell et al. 2008; Offringa et al. 2014).

Various algorithms to cope with such 3D problem have been implemented, several of them being enabled and optimized to exploit specific computational resources (e.g. GPUs or multicore platforms). Currently, the most widely adopted approach is the w-projection algorithm (Offringa et al. 2014), in which the w-term is expressed as a convolution in Fourier space and it is applied directly to the visibilities. Here w represents the third direction in the (u, v, w) coordinates system of the visibility space (see Section 2 for details). The w-projection algorithm has been parallelized and ported to the GPU by Lao et al. (2019), who however limited the data distribution to the visibility data. A CUDA GPU-based implementation is proposed also by Muscat (2014). An alternative approach is represented by the w-stacking technique (Cornwell et al. 2008), in which the data are partitioned in w, and the w-correction is applied in image space by multiplying each w-plane after FFT transform. This solution has been adopted by WSCLEAN (Offringa et al. 2014), followed by the work of Arras et al. (2021) and Ye et al. (2022), optimizing the original implementation. However, only multicore, shared memory architectures are supported. The WSCLEAN software implements also the IDG solution supporting both multicore and GPU implementations (van der Tol, Veenboer & Offringa 2018; Veenboer & Romein 2020). The gridding step has also been enabled to the GPU by Merry (2016). The w-stacking and w-projection approaches have been combined and parallelized on distributed memory systems by Pratley, Johnston-Hollitt & McEwen (2020). A further solution is represented by *faceting*, in which the w-correction is taken to be constant or linear over small regions of the sky (Cornwell & Perley 1992). The image is constructed by piecing together many different facets. For this approach, parallel multicore support can be found in the DDFACET package (Tasse et al. 2018).

The aforementioned solutions are only partially capable of fully facing the challenge related to big data exploiting modern, hybrid HPC solutions. In particular, parallel computing is not effectively used to process increasingly larger data sets and images, that cannot fit single memory systems, in conjunction with accelerated architectures that can dramatically reduce the time to solution.

In this work, we have addressed such challenge focusing on those steps of the imaging pipeline that characterize the w-stacking algorithm, namely gridding, FFT-transform, and w-correction. Hereafter, we refer to this combination of steps as w-stacking gridder. These steps are suitable to distributed memory parallelism, exploiting parallel FFT solutions and relying on a Cartesian 3D computational mesh that can be easily distributed and efficiently managed across different processing units, potentially leading to a good scalability on large HPC architectures.

Additionally, we have targeted portability as one of our prime goals. Given the large number of different HPC solutions currently available, and since the development of applications codes spans a period of time longer than the typical time-scale of HPC architecture evolution, our objective is to design a code easily portable on different computing platforms both in terms of code usability and in terms of performance. In this way, scientists can effectively exploit all the available supercomputing resources, without finding major obstacles in building and running the code on a new system, obtaining an acceptable performance and scalability anywhere.

The code has been developed adopting the C programming language standard (with extensions to C++ only to support GPUs through CUDA) and it has not been fine tuned to any specific computational architecture, in order to avoid introducing specialized solutions that would limit its portability. In order to have a portable solution even for the GPU implementation, alongside CUDA, we have experimented also the OpenMP support to offload operations to accelerators. We have not adopted performance portable solutions like OpenCL or Kokkos, since they introduce unwanted dependencies from additional libraries and applications.

We present the results obtained on two state-of-the-art supercomputing platforms: The Juwels Booster Module at the Jülich Supercomputing Centre (Germany, hereafter *Juwels*) and Marconi100 at the CINECA Supercomputing Centre (Italy, hereafter *M100*), that are ranked as third and sixth HPC systems in Europe in the TOP500 list[4] of 2022 June. Compilation on the two different platforms and with different compilers is done with a single source code version. The only required customization is related to compilation flags and environmental variables. All tests have been performed using LOFAR data sets, representative of current radio-observations.

The paper is organized as follows. The methods used for performing the w-stacking are described in Section 2. In Section 3, we describe the different HPC solutions adopted for the code. In Section 4, the results of the performance and scalability tests are presented and discussed. Conclusions are drawn in Section 5.

[3] https://basp-group.github.io/Puri-Psi/

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

2. THE *W*-STACKING GRIDDER

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

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

where u, v, w is a baseline coordinate in the coordinate system of the antennas and I is the spectral brightness and l, m is the sky coordinate. For small FoVs, the term $\sqrt{1-l^2-m^2}$ is close to 1, and equation (1) is an ordinary 2D Fourier transform, which, in order to speed-up the computation, is solved by using an FFT-based approach. This however, requires to map visibilities, that are point like data, to a regular mesh, which 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 FoVs are observed at once, visibility data from non-coplanar interferometric radio telescopes cannot be accurately imaged with a 2D Fourier transform: The imaging algorithm needs to account for the w-term. This term 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 (Cornwell et al. 2008), 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 phase correction is applied as

$$\frac{I(l,m)\,(w_{\max}-w_{\min})}{\sqrt{1-l^2-m^2}} = \int_{w_{\min}}^{w_{\max}} e^{2\pi i w(\sqrt{1-l^2-m^2}-1)} \times \iiint V(u,v,w) e^{2\pi i(ul+vm)} \mathrm{d}u\mathrm{d}v\mathrm{d}w. \quad (2)$$

For all the details, we refer to Offringa et al. (2014).

The w-stacking methodology has been implemented in a code using the C programming language, with some C++ extensions required by the CUDA GPU implementation, adopting a procedural programming approach. Schematically, the resulting code is shown in Fig. 1, in which we highlight the parts that have been subject to a distributed parallel implementation and those that have been also accelerated.

Two main data structures characterize the algorithm. The first is an unstructured data set storing the (u, v, w) coordinates of the antennas array baselines at each measurement time. Each baseline has a number of associated visibilities, which is determined by the frequency bandwidth, the frequency resolution and the number of polarizations. A further quantity, the weight, is also assigned to each measurement and polarization. The second, 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 the mesh. The two data structures determine the memory request of the algorithm. They are evenly distributed among different computing units as described in Section 3.1.

The code consists of five main algorithmic components, each supporting different types of HPC implementations. The first component takes care of reading observational data from binary files stored on the disc. The files are read in parallel, assigning the same fraction

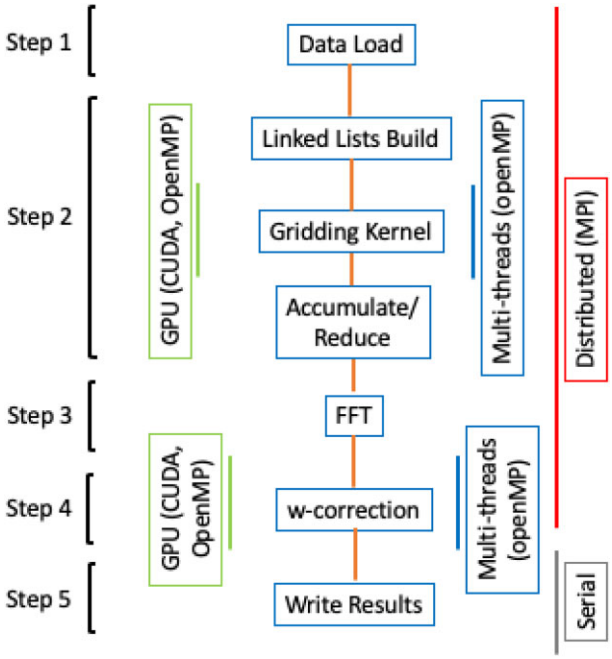

Figure 1 Schematic code architecture. Different kinds of HPC enabling are highlighted.

of the data to each different parallel task [message passing interface (MPI) task, see Section 3.1].

The following component performs the gridding of the visibilities. Gridding is done in successive rectangular slabs along the v-axis. The gridding procedure includes three substeps. In the first substep, a linked list is created for each slab, concatenating the data with u–v coordinates inside the corresponding slab. The second substep is represented by the gridding kernel. The linked list is traversed selecting the visibilities belonging to a given slab to convolve through a Gaussian kernel:

$$\tilde{V}(u_i,v_j,w_k) = \sum_{m\in\text{measures}} V_{\text{m}} G((u_m,v_m,w_m),(u_i,v_j,w_k)), \quad (3)$$

where m is the mth 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. The G kernel is a Gaussian convolution function with a kernel size of seven cells, following Offringa et al. (2014). Although Gaussian kernels are not used by any production imagers, leading to a high aliasing response in image space, in this work we have chosen to adopt such solution in order to include transcendental operations in calculation, supported differently by different computing architectures. Such operations would not be tested using typical production kernels, as, e.g. the Kaiser–Bessel function (Jackson et al. 1991), usually tabulated and interpolated. On the other hand, analytical approximations like Barnett, Magland & af Klinteberg (2019), are computationally equivalent to the Gaussian. In any case, different kernels are not expected to introduce substantially different computational overheads, as also stated by Offringa et al. (2014). The last substep is represented by the management of the data exchange among different computing units.

The third part of the algorithm performs the FFT of the gridded data, producing the real space image. This is performed using the FFTW library (https://www.fftw.org/).

The fourth step is the application of the phase shift and the reduction of the w-planes into the final image.

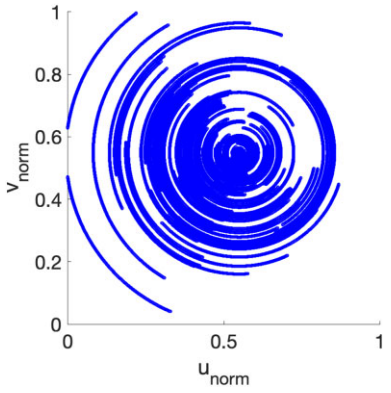

Figure 2 Example of measurements taken by a radio-interferometric telescope. An 8-h observation is shown. Both the angular coverage and the radial distribution can be significantly uneven. Both u and v coordinates are arbitrarily normalized.

In the fifth and final step the resulting image is written in a file on the disc.

3. THE HPC IMPLEMENTATION

The HPC implementation of our w-stacking gridder consists in: (i) distributing the data among different computing units, (ii) distributing the workload among different computing units, and (iii) speeding-up the work exploiting accelerators. Throughout the paper, 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 multithreaded OpenMP implementation, it is a multicore central processing unit (CPU). In the case of accelerated computing, it is a GPU.

3.1 Data distribution

In order to achieve an effective memory utilization, all big data sets, in our case those related to measured visibilities and those defined on the mesh, must be distributed. The concurrent presence of unstructured point like data (the observed visibilities), together with regular mesh based data (the convolved visibilities) require some care in order to obtain an efficient data distribution.

3.1.1 Measurement data distribution

Point like visibilities are collected as time series during an observation. At each measurement time (snapshot), data coming from all pairs of antennas of the interferometer (baselines) are collected. Snapshots are recorded one after the other. The resulting visibility distribution has a spatial number density that varies dramatically depending on the arrangement of the antennas and on the observation time. For instance, areas around $(u, v) = (0, 0)$ are denser than peripheral areas for arrays with a core-dense layout. Also radial number density is not uniform, changing with the orientation. Fig. 2 shows examples of such kind of uneven distributions.

In our approach, visibility data are distributed among processing units in time slices. If $t_{\rm obs}$ is the total observation time and $N_{\rm pu}$ is the number of processing units, we divide the data into $N_{\rm pu}$ time slices each comprising data in a time interval equal to $t_{\rm pu} = t_{\rm obs}/N_{\rm pu}$. Processing unit 0 owns the data from time 0 to $t_{\rm pu}$, processing unit 1 from $t_{\rm pu}$ to $2t_{\rm pu}$, and so on. An example of data distribution among four processing units is presented in Fig. 3.

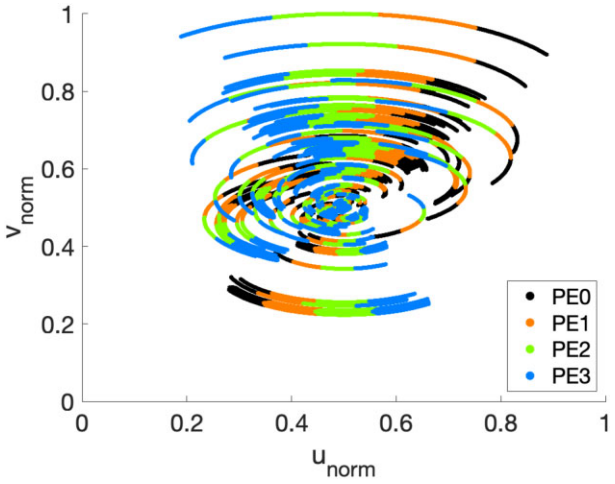

Figure 3 2D distribution of the observational data among four processing units (indicated as PE#). Subsequent time slices are assigned to each processor. Each processor owns data distributed on the whole (u, v) plane.

This data distribution has several advantages. First and foremost, data distribution is balanced, because each computing unit owns the same number of visibilities, which are spread on the whole (u, v, w) volume. Second, data can be distributed already when it is read from the disc, without subsequent communication. This, of course, requires also a parallel file system (which we assume to be available in an HPC facility) to be efficient. Third, data lying on the same area of the Cartesian grid where the visibilities are convolved are distributed among different computing units, reducing the computational pressure and potential race conditions (i.e. concurrent write access of multiple threads on the same grid point of the Cartesian mesh) on that specific area. Main drawbacks are represented by the non-locality of the data that are spread all over the spatial domain, hence the computational mesh, and possible unbalances for short observations. These issues could be alleviated, e.g. by sorting the data according to appropriate criteria. However, this would introduce additional computational and communication overheads, counterbalancing the possible benefits (see e.g. Hariri et al. 2016). Hence, we have chosen to adopt the linked-list-based approach described in Section 3.2.

3.1.2 Mesh data distribution

The data defined on the Cartesian computational grid has been partitioned among the computing elements adopting a rectangular slab-like decomposition (Fig. 4), 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_u and N_w are the mesh size in the u and in the w directions, respectively. N_v is the mesh size in the v direction, which is split in $N_{\rm pu}$ parts, equal to the number of computing tasks.

Such domain decomposition allows storing on each memory only a fraction of the whole mesh, avoiding replicas. Hence, each different computing element is in charge of storing and managing a specific part of the mesh (hereafter the *resident slab*). Furthermore, slab decomposition represents a natural layout for the usage of the parallel FFTW library, to transform mesh based visibility data to images. The main drawbacks of this approach are that (i) the maximum number of computing units that can be used is equal to the number of cells

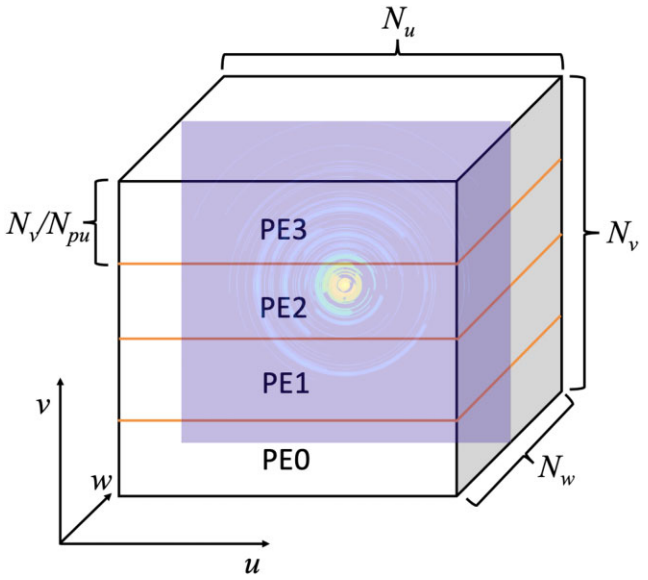

Figure 4 Cartesian computational mesh data distribution. Each computing unit (indicated as PE#) owns a rectangular *resident slab* of size $N_u \times N_w \times (N_v/N_{\rm pu})$ cells. In this example data are split among $N_{\rm pu} = 4$ computing units.

in the v-direction (along which the mesh is decomposed): $N_{\rm pu} \leq N_v$. Since, however, we deal with rather big $N_v > 1000$ meshes this is not expected to be a substantial limiting factor; and (ii) a suitable communication scheme has to be implemented in order to recompose the images.

3.2 Parallel processing

Data distribution implies additional management of the work performed by multiple computing units and some communication to communicate data among processors. Furthermore, synchronization has to be properly enforced in order to ensure that communication and work are properly completed at specific steps of code execution, when this is necessary (e.g. before writing results on disc). Communication and synchronization limit the effectiveness of the parallel execution of an algorithm, i.e. its scalability, which measures the ability to handle more work as the size of the computer or of the data grows. Therefore, they both have to be minimized in order to exploit larger and larger computing resources, supporting bigger data and reducing the time to solution. This is accomplished by exploiting the MPI (Gropp, Lusk & Skjellum 1994), the de-facto standard for distributed parallel computing.

3.2.1 Data I/O

Data read from disc is implemented by exploiting standard POSIX I/O on the top of a parallel file system. Each computing unit opens concurrently the input binary files and reads the assigned part of data (fseek, fread functions are used). Performance of the I/O subsystem has not been targeted by our work; therefore, no specific performance analysis and optimization have been carried out. The same holds for the writing operation of the final image. MPI I/O could be adopted to improve the performance of these operations if necessary.

3.2.2 Visibility mesh compositing

The Cartesian computational mesh is split into slabs, as described in Section 3.1 that are processed according to the following procedure.

Each processing unit concurrently calculates the local contribution of its own visibilities lying on a given slab. This local contribution is stored by each processing unit in an auxiliary buffer of the same size of the resident slab. As soon as visibilities of the given slab are processed, the local contribution stored onto the auxiliary slab is summed to the relevant resident slab on the processing unit to which it belongs to (resident slab i belongs to PEi)

The parallel summation is implemented adopting two different strategies: (i) using a blocking *MPI_Reduce* function from all the MPI tasks to that owning the resident slab, that implies that all communication has to be completed before the next iteration starts; and (ii) using a non-blocking one-side *MPI_Accumulate* operation, designed to perform concurrent reduce operations on the memory of a target processor. Synchronization is imposed only at the end of the loop over slabs, calling a collective *MPI_Win_fence* function.

This is performed for all the slabs in an iterative procedure, starting from the slab 0 up to the slab $N_{\rm pu-1}$. Once the last slab has been processed, each processing unit owns its up-to-date resident slab (i.e. with the contribution from the visibilities of the entire domain), ready for the following step (i.e. FFT). Fig. 5 sketches four compute units concurrently performing the gridding operation on their fraction of visibilities mapped on slab 2. Once the gridding has been performed, collective communications are used to accumulate or reduce the contribution of each processing unit on the resident slab on process 2.

It is worth highlighting that visibilities lying on the same slab (and belonging to the same processing unit) are scattered in memory, since there is no correlation between the memory location of a measurement and its physical spatial position, i.e. measurements that are close in memory can be spatially distant (and vice-versa). This can lead to a highly inefficient memory usage. In order to speed-up the memory access, prior to the loop over slabs, each processing unit creates $N_{\rm pu}$ linked lists. Each linked list files the local visibilities lying on a given slab. At each slab iteration, the corresponding list is traversed fetching only the data of interest. Fetched data are stored into temporary arrays that, being contiguous in memory, can be efficiently processed by the gridding kernel.

3.2.3 FFT and phase correction

The calculated resident slabs are processed through MPI parallel FFT, using the FFTW library. Each w-plane is computed separately. Finally, the w-term phase-shift correction is applied, summing all the different w-planes to generate the final 2D image. The phase correction is local and does not require any communication among MPI tasks.

3.3 Scaling laws

The scaling laws governing the parallel implementation of our code, can be modelled considering the critical model parameters, namely the number of visibilities $N_{\rm vis}$, the number of computing units $N_{\rm pu}$, and the Cartesian mesh size $N_{\rm I} = N_u \times N_v \times N_w$. In particular we focus on the parts of the code that have been subject to a distributed parallel implementation, namely, the gridding kernel, the FFT, the w-correction, and the parallel sum (indicated as 'reduction'). Each of those functions has a computational cost, in this context defined as the amount of time required to complete certain operation, C_i, that is a function of some of the model parameters.

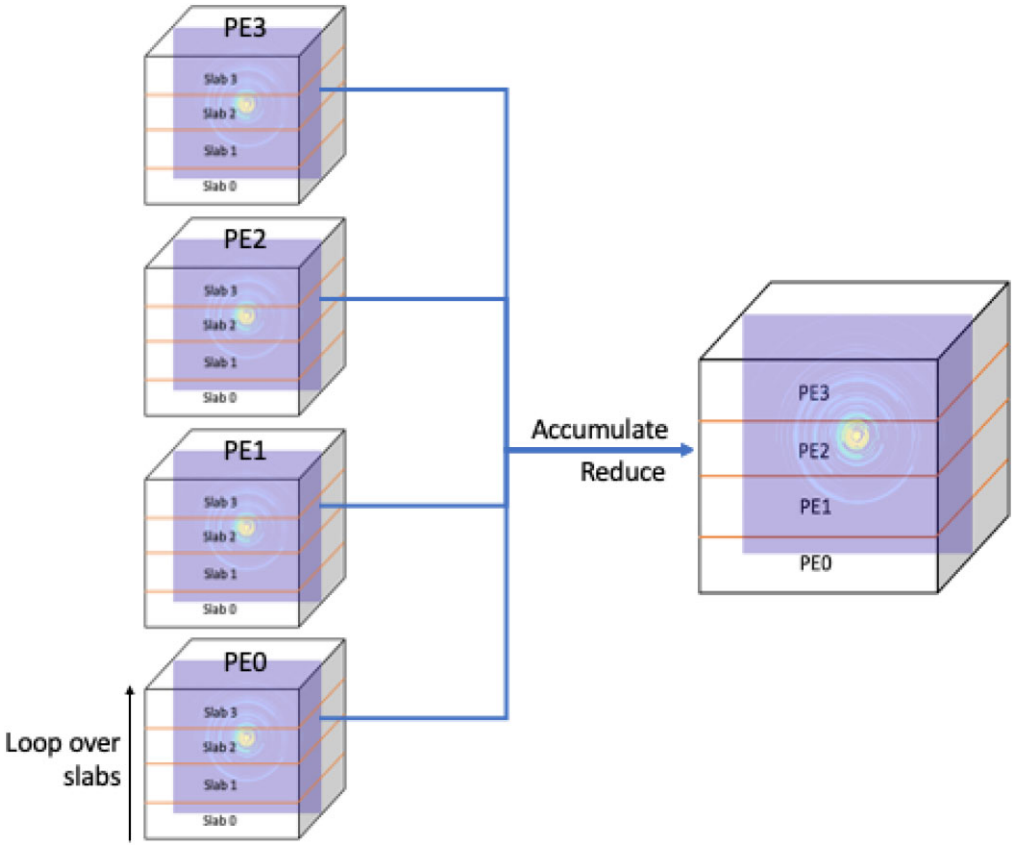

Figure 5 The different computing units (four units in this example, indicated as PE#) perform the gridding operation on their respective fraction of visibilities slab by slab. The process starts from slab 0: All processing units calculate their local contribution to slab 0 exploiting the linked list to accelerate visibility data access in memory. At the end, the contribution of each computing unit to slab 0 is *accumulated/reduced* (i.e. summed) on the resident slab, which is stored in PE0 memory. Then the process is repeated for slab 1 (accumulating on the resident slab on PE1), for slab 2 (as in this example, accumulating on the resident slab on PE2) and, finally, for slab 3 (accumulating on the resident slab on PE3).

The overall computational cost is

$$\begin{aligned} C(N_{\rm vis}, N_{\rm I}, N_{\rm pu}) = {} & \alpha_{\rm grid} \times C_{\rm grid}(N_{\rm vis}, N_{\rm pu}) \\ & + \alpha_{\rm FFT} \times C_{\rm FFT}(N_{\rm I}, N_{\rm pu}) \\ & + \alpha_{w} \times C_{w}(N_{\rm I}, N_{\rm pu}) \\ & + \alpha_{\rm red} \times C_{\rm red}(N_{\rm I}, N_{\rm pu}), \end{aligned} \tag{4}$$

where α_i are weights that do not depend on the model parameters.

The computational cost of the FFT is (Cooley & Tukey 1965)

$$C_{\rm FFT}(N_{\rm I}, N_w, N_{\rm pu}) \propto \frac{N_{\rm I}}{N_{\rm pu}} \log\left(\frac{N_{\rm I}}{N_{\rm pu} N_w}\right). \tag{5}$$

The w-correction cost is

$$C_w(N_{\rm I}, N_{\rm pu}) \propto \frac{N_{\rm I}}{N_{\rm pu}}. \tag{6}$$

Similarly, the cost of the gridding kernel is given by

$$C_{\rm grid}(N_{\rm vis}, N_{\rm pu}) \propto \frac{N_{\rm vis}}{N_{\rm pu}}. \tag{7}$$

It is worth noting that the parameter $\alpha_{\rm grid}$ depends on the square of the size of the smoothing kernel support (in our work, seven cells), that therefore has a relevant impact on the computational cost of the algorithm.

Finally the cost of the reduction operation is

$$C_{\rm red}(N_{\rm I}, N_{\rm pu}) \propto N_{\rm I} \times C_{\rm MPI}(N_{\rm pu}). \tag{8}$$

The linear dependency on the mesh size is due to the fact that in our implementation the total size of the communicated data in the reduction step does not depend on the number of processing units. The factor $C_{\rm MPI}(N_{\rm pu})$ depends on the summation algorithmic strategy (reduce versus accumulate), on the bandwidth, on the latency and the load of the network, and on the MPI implementation of the sum operation. Hence, its precise calculation depends on the local set-up and system usage conditions. It is anyway bigger than 1, progressively reducing the efficiency of the parallel sum with increasing the number of MPI tasks.

Overall, the computational cost depends linearly on the amount of observational data resident on each processing unit and linearly on the slab size. For a given problem size the corresponding contributions to the computational cost decrease with increasing the number of processing unit. On the contrary, the reduce term grows with $N_{\rm pu}$, hence its impact on the computational cost becomes progressively higher, possibly resulting dominant when a high number of MPI tasks is active and/or for small meshes and/or data sets.

3.4 Multi/many-threads parallelism

The time to solution for each single MPI task can be considerably reduced by accelerating the code exploiting multi-many-core so-

lutions. These solutions are represented either by multicore CPUs (multithreads parallelism) or by *accelerators* like GPUs (many threads parallelism). Accelerators can be adopted as general purpose co-processors coupled to CPUs to speed-up parts of the computation (*kernels*) that can be efficiently performed by independent tasks.

The gridding and the phase-correction algorithms have been targeted for multi/many threads parallelism. Convolution and *w*-correction of each measurement can in fact be performed independently from all the others with a high arithmetic intensity, representing an ideal case for multiple threads parallel execution. The main potential bottleneck is represented by the accumulation of the contributions of each measurement on the computational mesh that leads to frequent race conditions.

3.4.1 Multithreaded CPU implementation

Multithreading for multicore CPUs has been implemented by using the OpenMP[5] application programming interface. OpenMP represents a consolidated and stable solution to exploit shared memory devices, ensuring good scalability up to many cores and high portability, being supported by all the major hardware and software providers and implemented in all the most common compilers of, in particular, Fortran, C, and C++ programming languages. It is based on a set of directives which instruct the code on how to split the work among the available threads, with minimum impact on the source code, although specific customization has to be done in order to ensure good performance and scalability.

The OpenMP directives are ignored (interpreted as comments) by the compiler if OpenMP is not explicitly enabled at compile time (e.g. for the GCC compiler this is achieved by means of the `-fopenmp` compilation flag).

In our code, OpenMP is used to parallelize the gridding and phase-correction loops, the former over the measurements, and the latter over the grid cells. In both cases, loop iterations are independent from one another and they can be trivially distributed among OpenMP threads. When it comes to accumulation of the contributions from different threads, one has to take care about race conditions (i.e. several threads updating the same memory location simultaneously). We then use the OpenMP built-in *atomic* clause.

3.4.2 GPU implementation

The approach adopted for the GPU is similar to that described for multithreading.

For the gridding algorithm, each measurement is assigned to a different thread that performs the convolution of all frequencies and polarizations. Required data are contiguous in memory, hence effectively coalesced, leading to an efficient memory usage. Having millions of measurements a high occupancy of the GPU can be achieved.

In the case of the phase-correction kernel, each iteration of the nested loops in the three coordinate directions is assigned to a different thread.

The code implementation adopts two different approaches, the first based on CUDA,[6] the second on the offload constructs of OpenMP. The former is a programming model developed by NVIDIA for general computing on GPUs. It is currently the most effective and comprehensive approach to program graphic accelerators. However its usage is restricted to NVIDIA GPUs limiting the software portability. The latter consists in a set of directives introduced starting from OpenMP version 4.5, allowing developers to offload data and execution to target accelerators such as GPUs. The usage of a directive based approach hides some of the complexity intrinsic to a procedural approach like CUDA. The compiler is in charge of translating the kernels decorated with directives into constructs addressing the accelerator. This, at the expense of some control, which limits the maximum achievable performance. Furthermore, although in principle OpenMP is portable across different platforms and architectures, it is still supported by a limited number of compilers. GCC, version > 9, currently supports the full standard and it provides accelerated codes for NVIDIA and AMD GPUs. NVIDIA recently released an HPC standard development kit that includes C and Fortran compilers and a full support for a set of programming models including OpenMP, OpenACC, and CUDA.

CUDA implementation

In order to maintain a single source code and to preserve its portability, the sections of the code specific to the GPU implementation are activated at compile time exploiting the `__CUDACC__` macro that is automatically defined by the CUDA NVCC compiler.

If such macro is defined, the two functions targeting the GPU: (i) include the management of the data, copying to/from the GPU those (and only those, in order to minimize data transfers) arrays necessary for the computation; (ii) switch the loop over visibilities and those over the three coordinate directions to GPU kernel calls, assigning each iteration to a different thread; and (iii) manage the race conditions updating the shared arrays by means of the `atomicAdd` CUDA operation, which has hardware support to optimize the concurrent write-access to memory locations.

If not compiled with NVCC, the compiler ignores the CUDA sections and the code runs on the CPU.

OpenMP implementation

OpenMP directive-based programming model allows the users to insert non-executable `pragma` constructs that guide the compiler to handle the low-level complexities of the system.

Currently OpenMP offers a set of directives for offloading the execution of code regions onto accelerator devices. The directives can define target regions that are mapped to the target device, or define data movement between the host memory and the target device memory. The main directives we are implementing are `target data` and `target`, which create the data environment and offload the execution of a code region on an accelerator device, respectively. Both directives are paired with a map clause to manage the data associated with it; the map clause specifies the direction (to, from, or tofrom) of the data movement between the host memory and the target device memory.

In case of multiple GPUs available on the same node, a flag is used to map each GPU to a different MPI task, in this way we optimize the use of the available resources. The `__ACCOMP__` macro, is used to activate at compile time some OpenMP specific functions (e.g. inspecting the GPUs hardware and number of available GPUs).

Race conditions are addressed using the same *atomic* directive used for the CPU version.

The compiler used for all the OpenMP tests is the NVIDIA Standard development kit NVCC compiler, the same framework used for the CUDA tests.

4. RESULTS

In order to analyse the performance and the scalability of the HPC implementation of our code, we have performed a number of tests and benchmarks exploiting two different state-of-the-art

[5] https://www.openmp.org
[6] https://developer.nvidia.com/cuda-zone

Table 1. Technical characteristics of the two HPC platforms adopted for the tests presented in the paper, the adopted compilers and the corresponding compilation options.

System	Computing node	$N_{\rm nodes}$	Interconnect	Compiler	MPI distribution
M100	2×16-cores IBM Power9 AC922 CPUs + 4×NVIDIA Tesla V100 GPUs	980	Mellanox IB EDR DragonFly++	GCC 8.4.0 NVCC 10.1	Spectrum MPI 10.4.0
JUWELS	2× 24 cores AMD EPYC 7402 24C CPUs + 4×NVIDIA Tesla A100 GPUs	940	4×InfiniBand HDR (Connect-X6)	GCC 10.3.0 NVCC 11.3	ParaStationMPI 5.4.10

Table 2. Main characteristics of the LOFAR HBA observation used for the medium and large test data sets. Coordinates of the target are given in the first two rows.

LOFAR HBA inner station	
RA	15:58:18.96
DEC	+27.29.19.2
Observation time	8 h
Integration time	4 s
No. of antennas	62
Bandwidth	120–170 MHz
No. of sub-bands	25
No. of channels per sub-band	20
Sub-band bandwidth	2 MHz
MaxUVdist (wavelengths)	58624.99
MaxUVdist (m)	119902.23
MinUVdist (wavelengths)	13.61
MinUVdist (m)	27.83
FoV	∼12 deg^2
Angular resolution	∼6 arcsec

HPC architectures, available at Jülich and CINECA supercomputing centres.

The Forschungszentrum Jülich operates the Juwels, with 936 2×AMD EPYC 7402 24C 2.8GHz nodes equipped with 4 NVIDIA A100 GPU, 40 GB HBM2e each, connected by a 4×InfiniBand HDR interconnect. CINECA is the Italian national HPC facility running the M100 system, made of 980 2×16-cores IBM Power9 AC922 CPU nodes equipped with 4 NVIDIA Volta V100 GPUs. Each CPU node has 256 GB of DDR4 memory, the memory of the GPU is a 16 GB HBM2. The CPU and the GPU are interfaced by a PCIe Gen4 interconnect, while the four GPUs per node adopt NVLink 2.0. The system interconnect is a Mellanox IB EDR DragonFly++. Both systems deploy an IBM Spectrum Scale parallel file system.

Table 1 summarizes the characteristics of the systems and indicates the version of the adopted compilers and MPI libraries.

Two different input data sets have been used for the tests and the benchmarks, namely the *medium* and the *large* data set. The two data sets come from a LOFAR High Band Antenna (HBA) Inner Station 8-h observation, whose main characteristics are presented in Table 2. The medium data set is a single sub-band at 146 MHz with bandwidth of 2 MHz (such narrow bandwidth is peculiar of the specific observation, typical HBA observations having a 48-MHz bandwidth) split into 20 channels. It has been used for benchmarking the single node/GPU implementation of the code and for strong scalability tests. The large data set is made of the full observation, consisting of 25 sub-bands of 2 MHz each, spanning a frequency range between 120 and 170 MHz. It is used for weak scalability tests and to produce the 'large images' presented in Section 4.3. The main features of the two data sets together with the size of the computational mesh adopted in the two cases (increasingly bigger), are presented in Table 3.

Table 3. Characteristics of the input data sets and the computational mesh used in the medium and large tests. The mesh size takes into account the total amount of memory required for the real and imaginary part.

	Medium	Large
No. of visibilities (approx)	0.54×10^9	13.6×10^9
Input data size (GB)	4.4	102
Mesh size max ($N_u \times N_v \times N_w$)	$4096 \times 4096 \times 16$	$32\,768 \times 32\,768 \times 64$
Mesh size (GB)	4.0	1024.0

In all the systems the same source code has been compiled using the GCC and the NVCC compilers and the available MPI library. The only dependency of the code is on the FFTW3 library. The Makefile requires only few adjustments of several variables (e.g. the path to FFTW or to the CUDA libraries) to compile the code. The building procedure requires only a few seconds.

4.1 Scalability

We first analyse the scalability of our code, broadly defined as the ability to handle more work as the size of the computing system or of the application grows. Strong and weak scalability have been measured.

4.1.1 Strong scalability

A first set of tests focuses on the strong scalability of the code using the medium data set (see Table 3). Strong scalability measures the performance of the code keeping the data and the mesh size constant and progressively increasing the adopted computational resources (i.e. the number of MPI tasks, the number of OpenMP threads, and the number of GPUs). Ideally the code execution time should scale linearly with the number of computing units (i.e. the time should halves if the number of computing units doubles). However, various factors impact such an ideal behaviour, leading to a degradation of the performance with increasing the number of computing units. Therefore, strong scalability indicates the realistic gain the user can expect increasing the number of computing units working cooperatively.

The strong scalability of the whole code and of its main algorithmic component is measured by the *speed-up* parameter, calculated as

$$S = \frac{T_{A,1}}{T_{A,n}} \tag{9}$$

where $T_{A,\,n}$ is the wall clock time measured on system A using N computing units and $T_{A,\,1}$ is the corresponding time measured on the same system using a baseline configuration (e.g. a single computing unit). OpenMP tests are performed only on a single socket of a computing node (larger configurations being inefficient due to non-local memory accesses), while MPI and GPU tests are performed up to four nodes.

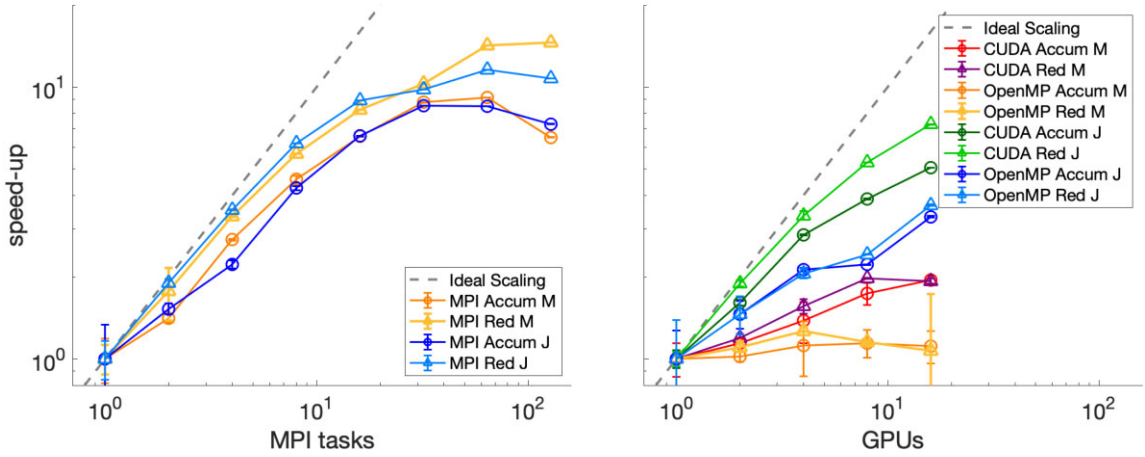

Figure 6 Strong scalability of the whole code on the different computing systems: M100 (M) and Juwels (J). Pure MPI runs are shown in the left-hand panel, GPU runs are shown in the right-hand panel. With Accum and Red, the MPI_Accumulate and MPI_Reduce versions are indicated, respectively. With CUDA and OpenMP the corresponding GPU implementations are indicated. Strong scaling is measured in terms of speed-up (see equation 9). Ideal linear scaling is shown as a grey, dashed line.

Strong scalability of the whole code on the CPU

The strong scalability of the whole code is presented in Fig. 6. For both the adopted computing systems, using the CPU cores (*pure MPI* runs, left-hand panel) an almost linear scalability is kept up to 32 cores. The speed-up decreases above such number, reflecting the increased impact of communication to the total computing time, due to several issues, discussed in Section 3.3. In summary: (i) the amount of communicated data does not change with increasing the number of MPI tasks, while the computational load of each task decreases (having to process smaller and smaller domains), (ii) the number of MPI operations increases with the number of tasks, with a bigger traffic overhead; in addition: (iii) above 32 cores communication involves the interconnect, which is anyway slower than local memory access, and (iv) unbalances in the code execution grow with the number of tasks, due to the number of data lying on a given slab, which can be different between the different processing units.

In order to alleviate the impact of unbalances, the MPI_Accumulate-based communication scheme has been introduced. Differently from the MPI_Reduce-based approach, which imposes synchronization at every iteration sweep over slabs, accumulating overheads due to unbalances, the MPI_Accumulate approach is non-blocking. It requires synchronization only at the end of the loop over slabs. Since the total number of visibilities is the same on each computing unit, this approach is expected to be unaffected by unbalances local to the slabs.

The scalability curves however show an opposite behaviour, with the MPI_Accumulate showing slightly worse scalability compared with the MPI_Reduce. This will be further discussed later in this section, taking into account also the outcomes of the weak scalability tests.

Strong scalability of the whole code on the GPU

When GPUs are used (right-hand panel of Fig. 6), scalability is influenced by the faster execution time of the GPUs compared with the CPUs, and the corresponding bigger impact of communication, that leads to a degradation of the speed-up (see Section 3.3). Scaling varies depending on the GPU implementation. The CUDA version shows a better scalability than the OpenMP one. The execution of the GPU kernels of the OpenMP version is, in fact, slightly faster than the CUDA one, leading to a higher impact of communication

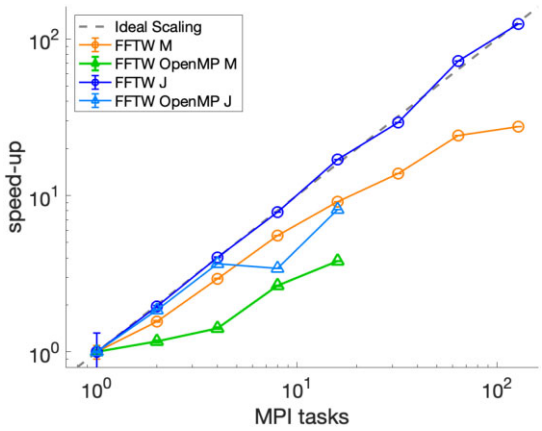

Figure 7 Strong scalability of the FFT algorithm on the different computing systems: M100 (M) and Juwels (J). The OpenMP CPU version of the code is indicated as OpenMP. The remaining curves correspond to pure MPI parallelism. Strong scaling is measured in terms of speed-up (see equation 9). Ideal linear scaling is shown as a grey, dashed line.

and to a consequent degradation of the scaling. The usage of the FFTW library has a non-negligible influence on the scalability of the GPU version. The fraction of the total computing time spent on the FFTW library is, in fact, relatively higher for the GPU version compared with the CPU one, since gridding and *w*-correction are performed much faster on the accelerator. On Juwels the pure MPI scalability of the FFTW library is linear, preserving a good scaling also for the whole application (see Fig. 7). The speed-up is instead sublinear on M100. In the two tests indicated as *OpenMP*, FFTW has been recompiled using the NVCC compiler, which produces a less scalable FFTW library.

Strong scalability of the gridding and *w*-stacking kernels

Figs 8 and 9 present the speed-up of those parts of the code that do not imply any communication, namely the gridding and the phase-correction kernels. For these kernels the work is evenly distributed among the computing units, as discussed in Section 3.3. The pure MPI multicore tests (left-hand panel in both figures), show a perfectly linear scaling in all systems. The central panel presents the speed-up

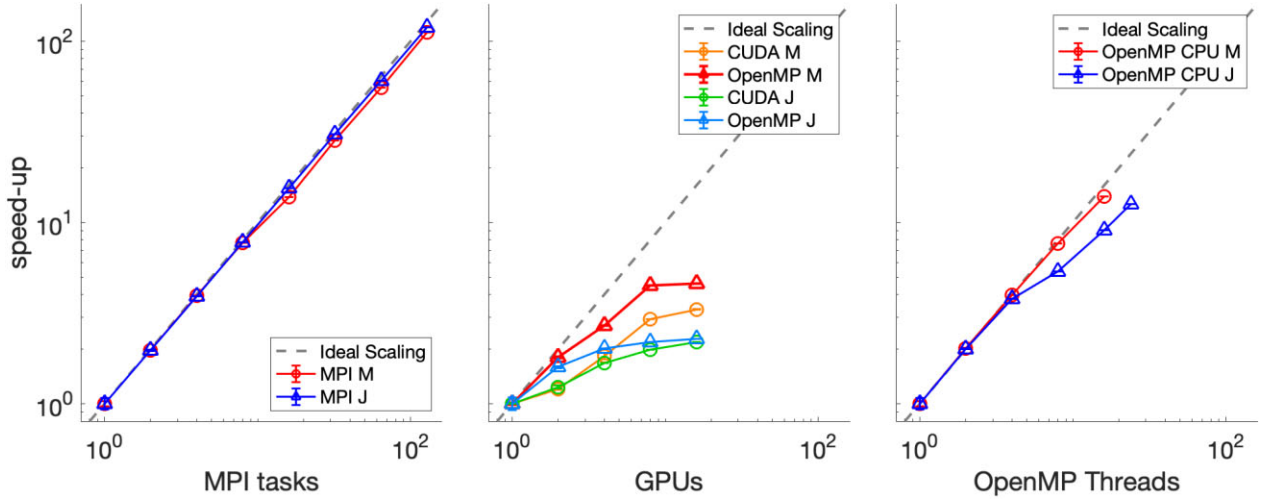

Figure 8 Strong scalability of the gridding algorithm on the different computing systems: M100 (M) and Juwels (J). Pure MPI runs are shown in the left-hand panel, GPU runs are shown in the centre panel (with CUDA and OpenMP the corresponding GPU implementations are indicated), and pure CPU OpenMP runs (limited to a single socket, 16 cores for M100, 18 cores for Juwels) are shown on the right-hand panel. Strong scaling is measured in terms of speed-up (see equation 9). Ideal linear scaling is shown as a grey, dashed line.

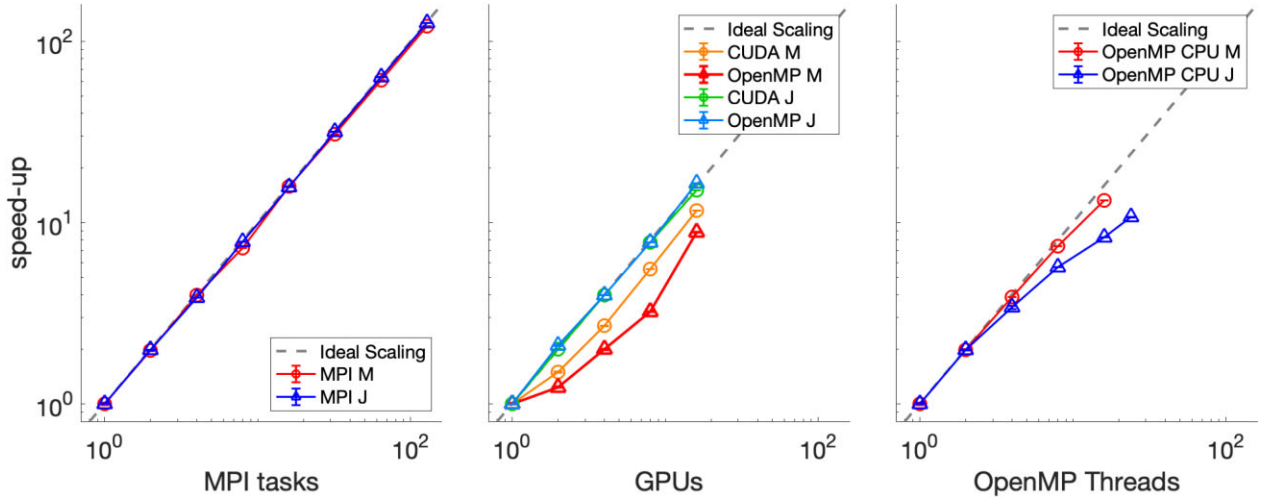

Figure 9 Strong scalability of the phase-correction algorithm on the different computing systems: M100 (M) and Juwels (J). Pure MPI runs are shown in the left-hand panel, GPU runs are shown in the centre panel (with CUDA and OpenMP the corresponding GPU implementations are indicated), and pure CPU OpenMP runs (limited to a single socket, 16 cores for M100, 18 cores for Juwels) are shown on the right-hand panel. Strong scaling is measured in terms of speed-up (see equation 9). Ideal linear scaling is shown as a grey, dashed line.

when GPUs are used. Nearly linear scaling is obtained by the phase-correction kernel, due to the perfectly data parallel algorithm. For the gridding algorithm, the GPU implementations do not approach a linear speed-up due to the progressive loss of efficiency of the GPU as the size of the amount of data to process decreases. This happens distributing the data among multiple accelerators. When 8 and 16 GPUs are used, the V100 architecture keeps increasing the performance (although sublinearly), while data are not big enough to lead to an efficient usage of the more powerful A100 GPU.

In the right-hand panel of the two figures, the OpenMP multicore scalability is presented. OpenMP data are presented up to the number of cores on a single socket, ensuring data locality through system memory-process bindings. The loop scheduling policy is static in order to exploit memory locality. For both kernels the resulting speed-up is close to ideal, although slightly worse than in the pure MPI case.

4.1.2 Weak scalability

Weak scalability has been studied using the large data set. In 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. Weak scaling is particularly meaningful for applications where the memory request cannot be satisfied by a single node, as happens for large observational data and/or large images, addressing high resolution and/or wide FoVs.

Fig. 10 shows the total (left-hand panel) and the FFTW time to solution normalized to the time measured on one node. Timings have been taken on a number of nodes equal to 1, 2, 4, and 8, corresponding to a number of cores (and MPI tasks) from 32 to 256 and a number of GPUs (and MPI tasks) from 4 to 32.

Input data correspond to the medium data set on one node, then they are doubled together with the number of nodes. The largest input

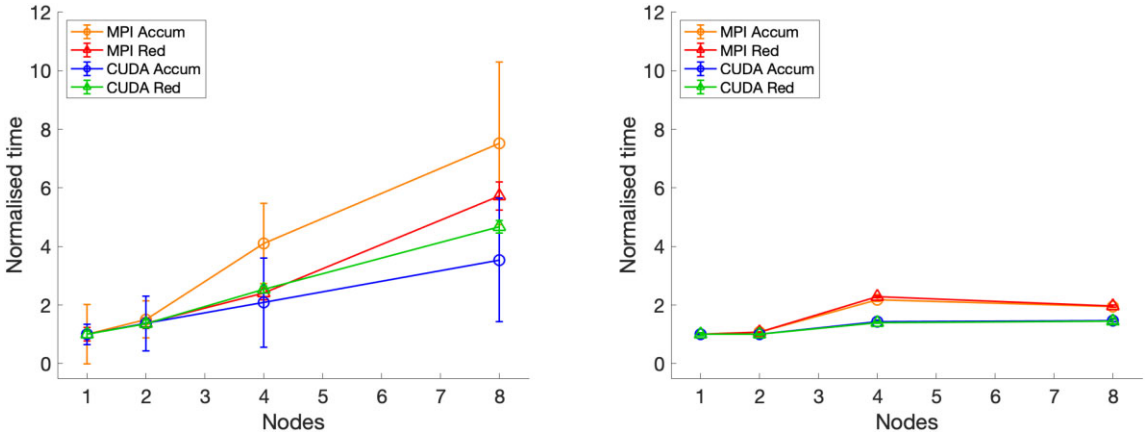

Figure 10 Weak scalability of the whole code (left-hand panel) and of the FFT algorithm (right-hand panel) on the M100 system. With Accum and Red, the MPI_Accumulate and MPI_Reduce versions are indicated, respectively. With MPI and CUDA, pure MPI runs and GPU runs are indicated. Wall-clock times are normalized to the time to solution on one node.

data set, distributed among eight nodes, is about 36 GB. At the same time, the size of the computational mesh is doubled starting from a $4096 \times 4096 \times 16$ set-up. The largest mesh is $8192 \times 8192 \times 32$ corresponding to a total memory occupancy of 32 GB.

With an ideal weak scaling, the time to solution should remain constant with the number of nodes. Left-hand panel of Fig. 10 shows that actually the measured total time to solution increases with the problem size and the number of nodes. Such growth is due essentially to MPI related communication overheads, as discussed in Section 3.3. The FFTW curves, presented in the right-hand panel, are substantially flat showing ideal scaling. The same holds for gridding and the phase-correction kernels (not shown here being perfectly constant). The overhead is therefore due to the gather of the slab data, and depends on the specific adopted communication approach.

When the MPI_Reduce collective communication function is used, the pure MPI case shows a larger overhead compared with the GPU case, due to the larger number of MPI tasks (eight times larger), as expected from equation (4). When the one-side MPI_Accumulate-based solution is used, such difference is even more visible. If few MPI task are used, as in the case of the GPU runs, the accumulate-based solution provides the best scaling, thanks to its asynchronous nature. However, for a number of tasks bigger than 64 (two nodes, for the pure MPI case) the scaling significantly worsen. Despite non-blocking, also the accumulate solution requires synchronization at the end of the loop over all the slabs. This is implemented through and MPI_Win_fence call, deployed by the MPI2 standard. Such a synchronization proves to introduce a strong overhead above 64 tasks, which appears to be a limiting factor in its usage on large configurations. It does not seem to be due to a specific implementation of the standard, affecting both the SpectrumMPI library available on M100 and the ParaStationMPI library available on Juwels.

4.2 Performance and memory usage

In this section, we discuss the performance of the code on a single computing unit. The wall-clock times to solution for the medium data set to generate a $4096 \times 4096 \times 16$ pixels image are presented in Table 4. This is the largest set-up that can run on a single GPU (due to memory constraints). Combined to the scalability information presented above, these figures characterize the computational performance of the discussed algorithm and assist the user in estimating the time needed for the *w*-stacking gridder within a data processing pipeline fully exploiting diverse and heterogeneous HPC resources.

4.2.1 Single core performance

The figures collected on a single core (first row of Table 4) inform on the time to solution when no parallel processing is adopted. The overall time for the application to run varies from around 87 s on Juwels to 130 s on M100. The Juwels system has higher performance compared with M100 in the gridding and phase-correction kernels (a factor of more than 2), given to the effective vectorization provided by the AMD processor architecture. The M100 machine deploys instead a highly efficient FFTW library (three times faster than the Juwels installation).

4.2.2 Single GPU performance

The CUDA implementation of the gridding and phase-correction kernels shows the expected speed-up compared with the single core benchmarks. All the timings of the CUDA implementation, presented in the second row of Table 4, include the data copy time to/from the accelerator. The gridding algorithm is 25 and 30 times faster than a single core on the M100 and Juwels systems, respectively. The phase-correction kernel has an even higher performance boost compared with a single core, with a factor of 75 for the V100 and a factor of 40 for the A100 architectures.

The FFTW library has no GPU implementation. Timings of the FFTW are the same as those of a single core, representing the highest share of the overall computing time. On the Juwels system, the FFTW built with the GCC compiler contributes to more than the 80 per cent of the total time. On M100 it contributes to the 55 per cent. Recompiling the FFTW library with the NVCC compiler and custom options improves its speed of a factor of 1.3 on M100 and of about 2 on Juwels (decreasing however its scalability as discussed in Section 4.1). The corresponding timings are presented in the third row of Table 4, where also the timings obtained using the GPU OpenMP implementation of the gridding and phase-correction kernels are shown (again, including data copy time to/from the GPU). It is interesting to notice that for our portable implementation, OpenMP results faster than CUDA on both computing systems. This proves that current offload capabilities of OpenMP can reach a good level of performance.

Table 4. Timings for processing the medium data set generating a 4096 × 4096 × 16 pixels image for the different code kernels on different devices (core, GPU, and node) and systems (M100 and Juwels). Errors are calculated as the standard deviation over 10 measurements. The last column shows the efficiency in terms of giga grid-point additions per second (GGPAS), i.e. the rate at which the cells of the computational mesh are calculated.

	System	Total (s)	Gridding (s)	Phase corr. (s)	FFTW (s)	GGPAS (GGPAS)
CORE	M100	130.80 ± 0.12	81.71 ± 0.03	36.732 ± 0.014	9.27 ± 0.08	0.328 ± 0.002
	Juwels	87.56 ± 0.16	36.47 ± 0.04	15.839 ± 0.004	31.62 ± 0.11	0.733 ± 0.006
GPU	M100	15.92 ± 0.03	3.25 ± 0.01	0.493 ± 0.005	9.27 ± 0.02	8.18 ± 0.02
CUDA	Juwels	36.79 ± 0.07	1.239 ± 0.004	0.397 ± 0.001	31.55 ± 0.06	20.84 ± 0.02
GPU	M100	12.11 ± 0.01	2.64 ± 0.02	0.309 ± 0.004	6.92 ± 0.03	10.09 ± 0.04
OMP	Juwels	18.56 ± 0.39	1.08 ± 0.08	0.400 ± 0.002	15.04 ± 0.01	26.15 ± 0.41
NODE	M100	12.74 ± 0.33	2.88 ± 0.01	1.193 ± 0.010	0.71 ± 0.03	8.79 ± 0.02
(cores)	Juwels	8.95 ± 0.16	1.19 ± 0.01	0.499 ± 0.001	1.22 ± 0.02	22.55 ± 0.04
NODE	M100	8.99 ± 0.23	1.79 ± 0.06	0.181 ± 0.01	3.17 ± 0.02	14.88 ± 0.18
(gpus cuda)	Juwels	10.92 ± 1.36	0.738 ± 0.005	0.098 ± 0.001	7.43 ± 1.38	36.11 ± 0.04
NODE	M100	9.58 ± 0.07	0.98 ± 0.02	0.158 ± 0.005	4.87 ± 0.03	27.06 ± 0.10
(gpus omp)	Juwels	9.00 ± 0.36	0.53 ± 0.03	0.100 ± 0.002	4.10 ± 0.02	49.60 ± 0.29

Overall, the presence of the non-accelerated FFTW, reduces the performance gain of the GPU enabled version of code to a factor of 8.5 for M100 and of slightly less than a factor of 2.5 on Juwels compared with a single core, emphasizing the importance of porting the whole code on the GPU to fully exploit the power of an accelerated system.

4.2.3 *Single node performance*

The analysis at the node level, measures the performance achievable by a software capable of exploiting all the computational resources of the available CPU. We present the timings obtained using all the cores on the node (pure MPI, fourth row of Table 4) together with those obtained using all the GPUs (four GPUs per node on both M100 and Juwels, presented in the fifth and sixth rows of Table 4 for the CUDA and the OpenMP implementations, respectively). For the gridding kernel, the CUDA and the OpenMP implementations are about 1.5 and 3 times faster than the pure MPI one, respectively, on M100, and 1.5 and 2.2 on Juwels. For the phase-correction kernel, we get factors of about 6.5 and 7 on M100 and of about 5 for both CUDA and OpenMP on Juwels.

Overall, on M100 the GPU implementation of the code is between 1.3 (OpenMP) and 1.4 (CUDA) times faster than the pure MPI one. On Juwels we get factors of about 0.8 and 1. However, the GPU code performance on the node is, once more, strongly influenced by the FFTW that cannot exploit the GPU.

The effectiveness of our implementation is expressed in terms of GGPAS, following the works published by Romein (2012) and Merry (2016). It is worth noting that GGPAS counts the number of updates in registers during the execution of the kernel, not the number of updates onto the grid points in the device global memory (i.e. GGPAS does not depend on the size of the grid). For the medium test the giga grid-point additions value is estimated to be equal to 26.65. The measured GGPAS follows of course the behaviour of the gridding kernel, with the best performance obtained by the GPU OpenMP implementation running on the Juwels system, with around 49.60 GGPAS. The GGPAS we get is lower than what achieved in the aforementioned works, where, however, specific optimizations have been accomplished. Although this could be done also for our code, it would be inconsistent with our goal of ensuring portability in terms of both code and performance, hence we have not pushed further the tuning of our implementation.

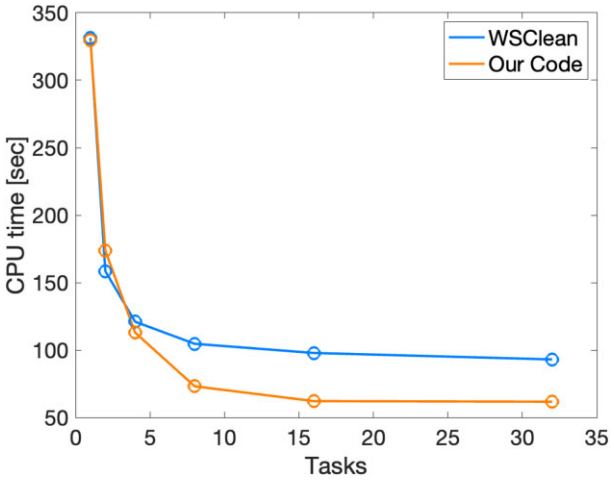

Figure 11 Comparison of the CPU time of WSCLEAN and the code presented in this paper ('our code') running on the same input data set with the same set-up, with increasing the number of parallel tasks on a single computing node. For WSCLEAN we have used multithreading while for our code we have adopted MPI parallelism. GPUs have not been used since they are not supported by version 3.0 of WSCLEAN. The same holds for multinode parallelism.

4.2.4 *Comparison with WSCLEAN*

Fig. 11 presents the comparison between the time to solution of our code and one of the state-of-the-art codes for imaging of radio astronomy data, WSCLEAN (Offringa et al. 2014). This comparison is meant only as a sanity check of our implementation and not a formal benchmarking, as it is possible that WSCLEAN performance can be improved with careful tuning of its parameters. However, this is outside the scope of this work. WSCLEAN, version 3.0, was run as follows:

wsclean -j N -size 4096 4096 -scale 2asec -name test input-data.mswhere the option -j sets the number of N threads that the

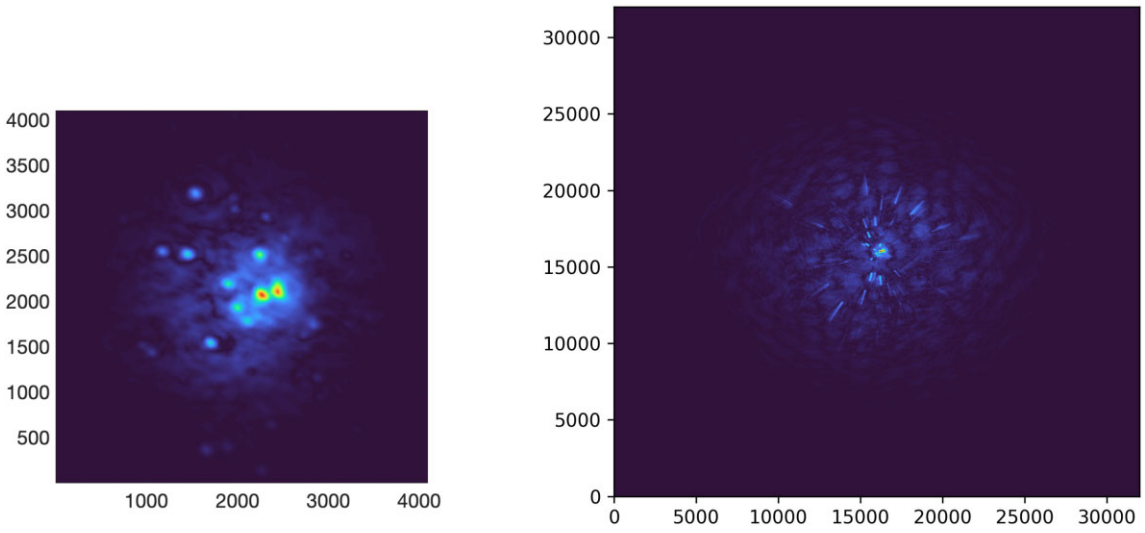

Figure 12 Result of the *w*-stacking procedure applied to the large data sets producing a 4096 × 4096 pixels (left-hand panel) and a 32 768 × 32 768 pixels (right-hand panel) image. The different mesh sizes reflect on the different FoVs of the two images. Smearing of outermost sources is due to the artificial combination of different bandwidths.

code can use, the -size option sets the image size in pixel, the -scale option sets the resolution in arcsec, and the -name option sets the prefix for the output files. With this set-up, WSCLEAN performs the same operations accomplished by our code. We have run only on one node and without using the GPUs, in order to have a fair comparison between the two codes. Built-in multithreading parallelism has been used for WSCLEAN, MPI based parallelism has been exploited for our code. We notice that the performance of the two codes is very similar up to four tasks. For larger configurations, multithreading scales a bit worse than MPI, leading to a slightly better performance of our code compared with WSCLEAN. Above 16 parallel tasks, both codes do not improve further the performance. For the MPI implementation this is due to the small size of the input data set, leading to overheads related to parallelism to dominate. Multithreading is instead penalized by a number of architectural issues (e.g. cache coherency, memory contention, etc.).

4.2.5 Memory requirements

The memory footprint of the *w*-stacking code per computing element (in bytes) can be calculated as

$$M \simeq 8 \times \frac{1}{N_{\rm pu}} \Big(3.5 \times N_{\rm b} + (2 \times N_{\rm vis} + 4 \times (N_u \times N_v \times N_w) + L_k \Big) \, {\rm B}, \tag{10}$$

where $N_{\rm b}$ is the number of baselines, $N_{\rm vis}$ is the total number of visibilities to be mapped to the grid, $N_u \times N_v \times N_w$ is the number of grid cells, and L_k is the size of the linked list assigning baselines to the different slabs. The factor 8 reflects the usage of double precision floating point numbers for most of the variables, the factor 3.5 is due to the three phase space coordinates of the baselines plus the single precision weight variable. The factor 2 takes into account the real and imaginary parts of the visibilities and the factor 4 is the product of the factor 2 for the real and imaginary parts of the visibilities calculated on the grid times a further factor of 2 due to the duplication of the slab to keep in memory both the resident slab and the slab used to convolve the visibilities. The size of L_k can be estimated as

$$L_k \sim 2 \times \alpha \times \frac{N_{\rm b}}{N_{\rm pu}}, \tag{11}$$

where the factor of 2 reflects the number of variables characterizing a node of the linked list (the id of the baseline and the pointer to the next node of the list), α is a parameter bigger than 1, that accounts for the boundary conditions for each slab (baselines falling in a neighbouring slab but contributing to that being processed). Such parameter depends from the size of the support of the convolution kernel (in our application equal to seven cells), but it is not expected to be >2. For $N_{\rm pu} > 2$, the contribution of L_k to M can be considered negligible, depending from the inverse of the square of the processing units.

Equation (10) shows that the size of all the main data structures involved in the computation scales linearly with the number of computing units, memory request halving with doubling the number of parallel tasks. The only meaningful memory overheads, compared with a purely sequential code, are represented by the additional resident slab and the linked list, that, however, has no impact starting from a few computing units.

4.3 Large images

Our final test is the processing of the large configuration reported in Table 3, with a mesh of 32 768 × 32 768 × 64 and an output image size of 32 768 × 32 768 pixels. In order to have the largest possible input data set, we have combined all the 25 bands of the available observation.

The test requires about 2.5 TB of memory, mapping to 128 nodes of the M100 system. Both pure MPI and GPU set-up have been run, the former using 4096 MPI tasks (32 cores per node), the latter 512 (four GPUs per node). In both cases the time to solution resulted to be between 2750 and 3150 s. In Fig. 12, the 32 768 × 32 768 pixels image is compared with a 4096 × 4096 pixels one. The latter is the minimum image size which ensures the nominal observation resolution (∼6 arcsec) to be sampled by at least 2 pixels. The increased resolution of the computational mesh of the former

reflects on its much larger FoV, the 4096 × 4096 field corresponding to a small square tile in the centre of the 32 768 × 32 768 image. The outermost sources are smeared due to the artificial combination of different bands, that would normally be treated separately and then properly merged, but that was required by our test to fulfil the computational requirements.

It is interesting to notice that the same memory requirements of the large test would be needed in order to process a 262 144 × 262 144 × 1 mesh. This means that a 262 144 × 262 144 pixels image can be generated with the above set-up.

5 CONCLUSIONS

In this paper, we have explored possible effective strategies towards the support of large radio-interferometric data sets, as those expected from the SKA radio telescope, by exploiting state-of-the-art HPC solutions. We have focussed on the imaging algorithm, specifically addressing the gridding, FFT transform, and *w*-correction steps, in terms of both memory request and computing time.

Our *w*-stacking gridder has been developed to demonstrate high performance and scalability on modern accelerated, multiprocessors devices, addressing also portability and maintainability, in order to facilitate the usage of the same software on different computing systems in a time scale much longer than that expected for a typical HPC technology, maximizing its usability and impact.

Our main achievements can be summarized as follows:

(i) We have enabled a prototype *w*-stacking gridder to the usage of heterogeneous HPC solutions, combining parallel computing with accelerators.

(ii) The resulting code supports data set whose size is limited only by the physical memory of the available computing units, evenly distributing data among distributed memories. Our proof of concepts run could perform the *w*-stacking of a 102 GB input data set on a 32 768 × 32 768 × 64 cells mesh *in memory* (overall memory request of about 2.5 TB) in <1 h.

(iii) The scalability of the code (both strong and weak) ensures that it can be efficiently used on large HPC configurations, involving hundreds or even thousands of cooperating computing units.

(iv) The usage the GPUs reduces the time to solution thanks to their outstanding performance, hence allowing using a smaller number of MPI tasks compared with a pure MPI set-up, alleviating communication related issues.

(v) Communication overheads can have a strong impact on the code performance when large images are generated using hundreds or thousands of computing units. However, the advantage of performing the full calculation in memory maintains the time to solution within reasonable limits (<1 h for our biggest case).

(vi) The code is easily portable on different systems, being written in standard C with some limited extensions to C++ and avoiding any tuning to a specific architecture. It can be compiled with a GCC GNU compiler and, in case NVIDIA GPUs are used, by the NVCC compiler. The only dependency, besides MPI, is from the FFTW library, which, however, can be easily installed on any computing system.

(vii) The usage of the offload support of OpenMP allowed achieving the porting to the GPU also adopting simple directives, in principle portable to different accelerated architectures, without (in this case) any performance penalty compared with CUDA. However, offload directives are fully supported only by GCC, version > 9, or NVCC version > 10.

(viii) The usage of HPC systems enables the processing of data and images of unprecedented size, particularly important for very large data sets and/or low frequency and very large baselines arrays.

This work represents only the first step towards a full exploitation of ultimate HPC solutions in radio astronomy. Additional steps are already taken in the direction of optimizing the usage of the GPUs (e.g. exploiting the CUDA FFT library, cuFFT, whose multinode version has been released at the beginning of 2022), which proved to deliver promising results in terms of performance, and extending the code to the exploitation of other kind of accelerators, in particular FPGA architectures. A critical step will be also the improvement of the performance related to communication both exploring novel patterns to minimize data exchange but also to acquire an in-depth comprehension of the MPI data transfer and synchronization mechanisms on a given interconnect, in order to exploit it at full efficiency. Finally, we will extend our study to include additional features towards the support of the full imaging procedure. More specifically we will introduce convolutional kernels commonly used in radio astronomy, as, e.g. the prolate spheroidal wave function or the Kaiser–Bessel window function (Jackson et al. 1991) and we will extend our parallel approach to the degridding procedure (transforming from the image to the visibility space). The same parallelization strategy is expected to be effective also for such inverse process, although a specific implementation will be required by the distribution of the visibilities on the cell back to the single measurements in the (*u*, *v*, *w*) space. Furthermore, we will support the full treatment of multiwavelength observations and of different polarization.

ACKNOWLEDGEMENTS

The authors want to thank Francesco De Gasperin, Luca Bruno, Luca tornatore, and Gianfranco Brunetti for the valuable discussions, advice, remarks, and suggestions that helped improving the contents of the paper. The High Performance Computing tests and benchmarks this work is based on, have been produced on the Marconi100 Supercomputer at CINECA Supercomputing Center (Bologna, Italy) in the framework of the ISCRA programme, IscrC_TRACRE project, and on the Juwels Booster Module at the Jülich Supercomputing Centre (Germany) thanks to the support by the European Union's Horizon 2020 Research and Innovation Programme under the EuroEXA project (Grant Agreement No. 754337). The data to perform all the tests have been kindly provided by the LOFAR project LC14_018, PI Franco Vazza.

DATA AVAILABILITY

The data used for this work have been produced within the LOFAR project LC14_018 and will be available according to the standard LOFAR policy.

REFERENCES

Arras P., Reinecke M., Westermann R., Enßlin T. A., 2021, A&A, 646, A58
Barnett A. H., Magland J. F., af Klinteberg L., 2019, SIAM J. Sci. Comput., 41, C479
Bhatnagar S., Cornwell T. J., 2017, AJ, 154, 197
Bhatnagar S., Cornwell T. J., Golap K., Uson J. M., 2008, A&A, 487, 419
Bhatnagar S., Rau U., Golap K., 2013, ApJ, 770, 91
Carrillo R. E., McEwen J. D., Wiaux Y., 2012, MNRAS, 426, 1223
Carrillo R., McEwen J., Wiaux Y., 2014, MNRAS, 439, 3591
Clark B., 1980, A&A, 89, 377

Cooley J. W., Tukey J. W., 1965, Math. Comput., 19, 297
Cornwell T., 1999, in Taylor G., Carilli C., Perley R., eds, ASP Conf. Ser. Vol. 180, Synthesis Imaging in Radio Astronomy II. Astron. Soc. Pac., San Francisco, p. 151
Cornwell T. J., Evans K. F., 1985, A&A, 143, 77
Cornwell T. J., Perley R. A., 1992, A&A, 261, 353
Cornwell T. J., Golap K., Bhatnagar S., 2008, IEEE J. Sel. Top. Signal Process., 2, 647
Dabbech A., Ferrari C., Mary D., Slezak E., Smirnov O., Kenyon J. S., 2015, A&A, 576, A7
Dabbech A., Repetti A., Perley R. A., Smirnov O. M., Wiaux Y., 2021, MNRAS, 506, 4855
Dabbech A., Terris M., Jackson A., Ramatsoku M., Smirnov O. M., Wiaux Y., 2022, ApJ, 939, L4
Girard J. N., Garsden H., Starck J. L., Corbel S., Woiselle A., Tasse C., McKean J. P., Bobin J., 2015, J. Instrum., 10, C08013
Greisen E. W., 2003, AIPS, the VLA, and the VLBA. Springer, Netherlands, p. 109
Gropp W., Lusk E., Skjellum A., 1994, Using MPI: Portable Parallel Programming with the Message-Passing Interface. MIT Press, Cambridge, MA
Hamaker J. P., Bregman J. D., Sault R. J., 1996, A&AS, 117, 137
Hancock P. J., Murphy T., Gaensler B. M., Hopkins A., Curran J. R., 2012, MNRAS, 422, 1812
Hariri F. et al., 2016, Comp. Phys. Commun., 207, 69
Hoekstra H., Hsieh B. C., Yee H. K. C., Lin H., Gladders M. D., 2005, ApJ, 635, 73
Högbom J. A., 1974, A&AS, 15, 417
Jackson J., Meyer C., Nishimura D., Macovski A., 1991, IEEE Trans. Med. Imaging, 10, 473
Jagannathan P., Bhatnagar S., Rau U., Taylor A. R., 2017, AJ, 154, 56
Johnston S. et al., 2007, Publ. Astron. Soc. Aust., 24, 174
Jonas J., MeerKAT Team, 2016, Proc. Sci., MeerKAT Science: On the Pathway to the SKA. SISSA, Trieste, PoS#001
Junklewitz H., Bell M. R., Selig M., Enßlin T. A., 2016, A&A, 586, A76
Kenyon J., Smirnov O., Grobler T., Perkins S., 2018, MNRAS, 478, 2399
Lao B., An T., Yu A., Zhang W., Wang J., Guo Q., Guo S., Wu X., 2019, Sci. Bull., 64, 586
McMullin J. P., Waters B., Schiebel D., Young W., Golap K., 2007, in Shaw R. A., Hill F., Bell D. J., eds, ASP Conf. Ser. Vol. 376, Astronomical Data Analysis Software and Systems XVI. Astron. Soc. Pac., San Francisco, p. 127
Merry B., 2016, Astron. Comput., 16, 140
Mitchell D. et al., 2010, Proc. Sci., RFI Mitigation Workshop. SISSA, Trieste, PoS#16
Mostert R. I. J. et al., 2021, A&A, 645, A89
Muscat D., 2014, Masters Thesis read at the University of Malta, preprint (arXiv:1403.4209)
Offringa A. R. et al., 2014, MNRAS, 444, 606
Onose A., Carrillo R. E., Repetti A., McEwen J. D., Thiran J.-P., Pesquet J.-C., Wiaux Y., 2016, MNRAS, 462, 4314
Pratley L., McEwen J. D., d'Avezac M., Carrillo R. E., Onose A., Wiaux Y., 2018, MNRAS, 473, 1038
Pratley L., Johnston-Hollitt M., McEwen J. D., 2020, Publ. Astron. Soc. Aust., 37, e041
Rau U., Cornwell T. J., 2011, A&A, 532, A71
Rau U., Bhatnagar S., Voronkov M. A., Cornwell T. J., 2009, IEEE Proc., 97, 1472
Repetti A., Birdi J., Dabbech A., Wiaux Y., 2017, MNRAS, 470, 3981
Romein J. W., 2012, Proc. 26th ACM Int. Conf. Supercomput. ICS '12, An Efficient Work-distribution Strategy for Gridding Radio-telescope Data on GPUs. Assoc. Comput. Mach., NY, USA, p. 321
Roscani V., Tozza S., Castellano M., Merlin E., Ottaviani D., Falcone M., Fontana A., 2020, A&A, 643, A43
Sault R. J., Teuben P. J., Wright M. C. H., 1995, in Shaw R. A., Payne H. E., Hayes J. J. E., eds, ASP Conf. Ser. Vol. 77, Astronomical Data Analysis Software and Systems IV. Astron. Soc. Pac., San Francisco, p. 433
Schwab F., 1984, AJ, 89, 1076
Serra P. et al., 2015, MNRAS, 448, 1922
Smirnov O. M., 2011a, A&A, 527, A106
Smirnov O. M., 2011b, A&A, 527, A107
Tasse C., van der Tol S., van Zwieten J., van Diepen G., Bhatnagar S., 2013, A&A, 553, A105
Tasse C. et al., 2018, A&A, 611, A87
Terris M., Dabbech A., Tang C., Wiaux Y., 2022, MNRAS, 518, 604
Thouvenin P.-A., Abdulaziz A., Dabbech A., Repetti A., Wiaux Y., 2020, preprint (arXiv:2003.07358)
Thouvenin P.-A., Dabbech A., Jiang M., Abdulaziz A., Thiran J.-P., Jackson A., Wiaux Y., 2022, MNRAS, preprint (arXiv:2209.07604)
van der Tol S., Veenboer B., Offringa A. R., 2018, A&A, 616, A27
van Diepen G., Dijkema T. J., Offringa A., 2018, Astrophysics Source Code Library, record ascl:1804.003
van Haarlem M. P. et al., 2013, A&A, 556, A2
Veenboer B., Romein J. W., 2020, Astron. Comput., 32, 100386
Wieringa M., Raja W., Ord S., 2020, in Pizzo R., Deul E. R., Mol J. D., de Plaa J., Verkouter H., eds, ASP Conf. Ser. Vol. 527, Astronomical Data Analysis Software and Systems XXIX. Astron. Soc. Pac., San Francisco, p. 591
Ye H., Gull S. F., Tan S. M., Nikolic B., 2022, MNRAS, 510, 4110

This paper has been typeset from a TEX/LATEX file prepared by the author.