FLUPS - a flexible and performant massively parallel Fourier transform library

Pierre Balty, Philippe Chatelain, and Thomas Gillis

Abstract—**Massively parallel Fourier transforms are widely used in computational sciences, and specifically in computational fluid dynamics which involves unbounded Poisson problems. In practice the latter is usually the most time-consuming operation due to its inescapable all-to-all communication pattern. The original `flups` library tackles that issue with an implementation of the distributed Fourier transform tailor-made for successive resolutions of unbounded Poisson problems. However the proposed implementation lacks of flexibility as it only supports cell-centered data layout and features a plain communication strategy. This work extends the library along two directions. First, `flups`' implementation is generalized to support a node-centered data layout. Second, three distinct approaches are provided to handle the communications: one all-to-all, and two non-blocking implementations relying on manual packing and `MPI_Datatype` to communicate over the network. The proposed software is validated against analytical solutions for unbounded, semi-unbounded, and periodic domains. The performance of the approaches is then compared against `accFFT`, another distributed FFT implementation, using a periodic case. Finally the performance metrics of each implementation are analyzed and detailed on various top-tier European facilities up to $49,152$ cores. This work brings `flups` up to a fully production-ready and performant distributed FFT library, featuring all the possible types of FFTs and with flexibility in the data-layout. The code is available under a BSD-3 license at github.com/vortexlab-uclouvain/flups.**

Index Terms—**Distributed Applications, Fast Fourier transforms,**

I. INTRODUCTION

The distributed implementation of the three-dimensional (3D) Fourier transform, or equivalently the successive application of three one-dimensional (1D) transforms, has been a computational challenge over the past decades. Encouraged by its very broad impact many works have been proposed recently on the parallel implementation of the Fourier transform, with a particular emphasis on GPU support. Among them, `accFFT`[2] is the fastest on both CPU-only and heterogeneous architectures[3]. Relying on a hybrid MPI and CUDA parallelism, it optimizes the overlap between communications and computations to hide the communication overhead. It also provides support for real-to-complex and complex-to-complex transforms. Similarly `heFFTe`[4] reaches good performance while removing some of the restrictions applied on the topology of the input data. To hide the transpose of the local data in the communications, *Dalcin et al.*[5] proposed an approach based on MPI datatypes and generalized all-to-all communications. All of the mentioned libraries have been thoroughly benchmarked and compared to other software, such as P3DFFT[6] or SWFFT[7], more details can be found in [3].

The applications of the Fourier transform are numerous, and in particular when solving PDEs in fields such as fluids dynamics or gravitational problems. In the specific field of computational fluid dynamics, incompressibility is accounted for through a Poisson equation which is often solved using an FFT-based Poisson solver [8–10]. Despite its relevance, a software with enough flexibility to be used in practice while retaining its parallel performance at large scale is still missing, hence forcing the users to write their own, usually not optimized implementations.

Based on our computational fluid dynamics expertise, we identify three main requirements for a distributed implementation of the Fourier transform to be convenient:

1) the combination of various FFT types must be supported and efficient. We here refer to real-to-real, real-to-complex, and complex-to-complex FFTs, as many PDE simulations rely on real numbers;
2) the user-provided data should be given either in a cell-centered or a node-centered data layout, as defined in section II-A;
3) the required flexibility on the user side must not compromise the parallel performance, both measured by scalability metrics and the time-to-solution.

The original `flups` library[1] focuses on the first and partially on the third requirement while omitting the second one. The authors proposed a software for the resolution of unbounded Poisson equations in 3D on uniform distributed grids with various boundary conditions (BCs) and all their combinations. They sort the different BCs into 4 categories: (1) the even and odd BCs corresponding to DST/DCT, (2) semi-unbounded BCs using extra padding and a DCT/DST, (3) periodic BCs via the DFT, and (4) unbounded BCs through DFT and extra padding. To reduce the computational cost, `flups` reorders the transforms according to the cost entailed by the BC type, starting with the most affordable ones. It thus follows the order: (1), (2), (3), and finally (4). However the implementation does not support the node-centered data layout and offers unsatisfying time-to-solution, especially with unconventional distribution of unknowns.

As most distributed FFT libraries, `flups` uses `fftw`[11] as a 1D FFT transform library and implements the various memory transfers and communications between distributed resources[12]. The latter drives the overall performance and is considered as the algorithm bottleneck, especially at large scale [13]. Two approaches stand out for the distribution of the 3D data among the resources [6], which consequently impacts the application of the FFTs as well as the communication strategies: the slab or the pencil decomposition. Both strategies distribute a 3D computational grid, for simplicity assumed here of size N in each dimension (hence with a total size of N^3), onto P computational resources. The slab

decomposition consists in dividing the grid into slices of data and performing 2D FFTs in the first two dimensions. It is followed by an all-to-all communication to reorder the data in the third direction and perform the remaining 1D FFT. The scalability of this method breaks down when the number of processes P exceeds the data size in the third direction N, as some of the processes do not own any data hence reducing the load balancing. On the other side, the pencil decomposition approach removes this partition-size limitation. For each of the three dimensions, it divides the data into 1D pencils, which leads to up to N^2 independent data chunks to be distributed among the processes. With that strategy 1D FFTs are performed in the pencil direction, and communications are needed to switch from one direction to another (and realign the data accordingly) between successive FFTs. While the slab decomposition requires each MPI rank to communicate once with all the other ranks in the communicator, which makes it efficient on small partitions, the pencil decomposition requires each MPI rank to communicate twice with $\sqrt{P}$, which reduces the communication cost on very large partitions.

To address the gap between the research efforts in computational science and the end-user applications, we propose a massively distributed FFT implementation that checks the three requirements we have identified: (1) mixed FFT types, (2) agnostic to the data layout, and (3) large scale scalability and performance. This work builds upon the existing `flups` library which has been used in computational fluid dynamics codes to solve unbounded Poisson problems. Therefore our motivation as well as our presentation of the methodology and the obtained results is driven by this specific application. However the performance results are not specific to this configuration and we expect the impact of our work to go beyond the application envisioned.

First we present in section II the generalization of the implementation to be compatible with both node-centered and cell-centered applications. Then in section III we detail our communication strategies, as well as several optimizations to improve scalability and peformance. The resulting codebase is validated against analytical solutions of the Poisson problem in section IV. In section V, we demonstrate the convergence of `flups` and its ease of use by solving the Biot-Savart equation, a variation of the Poisson problems already considered. Then, to ensure that the proposed changes and generalizations do not affect the performance, we compare our time-to-solution against `accFFT` in section VI. Finally, we assess the parallel performance of each communication strategy and benchmark the code on three top-tier European clusters. To conclude this work, we present a summary of the proposed innovations as well as our results in section VII.

II. METHODOLOGY

As an FFT-based approach, both the original and the proposed updated version of `flups`, solve the Poisson equation $\nabla^2 u = f$ through a convolution between the right-hand side f and a pre-computed Green's function G,

$$\nabla^2 u = f \Rightarrow u = G * f \ . \tag{1}$$

The latter being performed as a point-wise multiplication in the spectral space: $\hat{u} = \hat{G} \cdot \hat{f}$. The choice of the Green's function G as well as its spectral representation $\hat{G}$ depends on the boundary conditions (BCs) and the chosen regularization at the origin. The forward and backward multidimensional Fourier transform is performed through a succession of 1D FFTs, whose types are chosen to comply with the required boundary conditions (BC), as described in Section I. We refer the reader to [1, 14] for further details on the algorithm as well as the combination of unbounded and spectral directions.

As already mentioned `flups` has been designed for cell-centered data layout exclusively. To accommodate node-centered data layouts some parts of the algorithms have been generalized and/or rewritten. In section II-A we first detail the notation regarding the length of the domain in order to handle the two proposed data layouts. Then in section II-B we detail the choice of FFTs for the periodic, even, and odd boundary conditions, and in section II-C we finally detail the implementation of the (semi-)unbounded boundary condition.

A. Point numbering conventions

The presented library relies on `fftw` to take care of the FFT computations. For performance reasons, the authors of `fftw` adapt the length of the provided data to avoid any trivially null computations. As `flups` combines different types of FFTs and supports different boundary conditions, we have chosen to use the same convention for all the different cases (regardless of the type of FFTs):

As commonly defined in PDE simulations, a multidimensional grid with a physical size of L_i contains N_i data in the ith dimension (e.g. in 3D: $L_x \times L_y \times L_z$ with $N_x \times N_y \times N_z$ data). The data might be organized either in a node-centered or cell-centered data layout, defined for the ith dimension as:

- cell-centred layout: $f_j \triangleq f(x_j)$ where $x_j = (j + 1/2) h$ with $j \in [0 \ ; N-1]$ and $h = L/N$.
- node-centred layout: $f_j \triangleq f(x_j)$ where $x_j = (j) h$ with $j \in [0 \ ; N]$ and $h = L/N$.

We note that when considering the node-centered configuration the data size is $N + 1$, instead of N for the cell-centered layout. Specifically to the node-centered layout, the last point f_N is crucial in some configurations such as the even boundary condition. However the information is sometimes duplicated when considering periodic boundary conditions. In the latter case, the boundary points (0 and N) must match the imposed boundary conditions and might be used (or not) by `flups` depending on the chosen FFT. Nevertheless for an improved usability we ensure that those duplicated points contain the correct information in the end result. Finally, as the rest of this section relates to 1D definitions, we simplify the notation by using N instead of N_i and L for L_i when appropriate.

B. Periodic, even, and odd boundary conditions

For both data layouts, the periodic boundary condition relies on the 1-D real-to-complex DFT defined as

$$\tilde{f}_k = \sum_{j=0}^{N-1} f(x_j) \exp\left[-\mathrm{i} \frac{2\pi}{N} j k\right] = \sum_{j=0}^{N-1} f(x_j) \exp\left[-\mathrm{i}\, \omega_k x_j\right] \ , \tag{2}$$

with $i = \sqrt{-1}$, $k \in [0 \, ; \, N/2]$ and $\omega_k = k\frac{2\pi}{L}$ the frequency associated to the output k. Note that we use $\tilde{f}$ instead of $\hat{f}$ to distinguish the partially spectral result obtained after a 1D DFT from the fully spectral output obtained through the 3D FFT.

The real-to-complex DFT produces $N/2 + 1$ complex modes where the mode 0 and π (constant and flip-flop modes respectively) are purely real. As we use the standard complex storage of `fftw`, we don't explicitly take advantage of the trivially null imaginary parts. In the case of node-centered information, the last data provided by the user (f_N) is unused.

The choice of FFTs gets more diverse when considering real-to-real transforms used to impose even and odd boundary conditions. The different combinations of the forward transform are summarized in Table I for both cell-centered and node-centered layouts. For the cell-centered one we refer the reader to [1], and we detail hereunder the node-centered layout.

	node-centered	cell-centered
odd-odd	type-I DST	type-II DST
odd-even	type-III DST	type-IV DST
even-odd	type-III DCT	type-IV DCT
even-even	type-I DCT	type-II DCT

Table I: Spectral boundary condition and the corresponding forward Fourier transforms

1) Even-even boundary condition: The even-even condition is imposed using a type-I DCT, which contains the real part of the DFT as given in Equation (2). This DCT is then defined as

$$\tilde{f}_k = f(x_0) + (-1)^k f(x_N) + 2 \sum_{j=1}^{N-1} f(x_j) \cos\left[\frac{2\pi}{2N} j \, k\right] , \quad (3)$$

with $k \in [0 \, ; \, N]$. We note that this transform produces $N+1$ real spectral information corresponding to the frequencies $\omega_k = \frac{2\pi}{2L} k = \frac{2\pi}{L} \frac{k}{2} = \frac{\pi}{L} k$. The output of this transform is then equivalent to the frequencies obtained by using the DFT on a domain $2N$. In order to improve consistency across the definitions and in the implementation, we will use the notation based on $\frac{2\pi}{2L}$ throughout this section. As illustrated in Figure 1 where we highlight the needed information from the user point of view, both the values in f_0 and f_N are relevant in the even-even case.

2) Odd-odd boundary condition: To impose an odd-odd boundary condition, also illustrated in Figure 1, we use the type-I DST which contains the imaginary part of the DFT as given in Equation (2):

$$\tilde{f}_k = 2 \sum_{j=1}^{N-1} f(x_j) \sin\left[\frac{2\pi}{2N} j \, k\right] , \quad (4)$$

with $k \in [1 \, ; \, N-1]$, where $\tilde{f}_0 = 0$ and $\tilde{f}_N = 0$ since the input data are real. To match this definition the first and last user-provided data are assumed to be zero and `fftw` discards both information to reduce the memory footprint and the time-to-solution. To adapt to this convention in `flups` we consider u_i for $i \in [1 \, ; \, N-1]$ only and the information located in $i = 0$ and $i = N$ are overwritten to be zero.

3) Odd-even boundary condition: In order to impose mixed boundary conditions such as the odd-even one we

use a type-III DST as defined by

$$\tilde{f}_k = (-1)^k f(x_N) + 2 \sum_{j=1}^{N-1} f(x_j) \sin\left[\frac{2\pi}{2N} j \left(k + \frac{1}{2}\right)\right] , \quad (5)$$

with $k \in [0 \, ; \, N-1]$. Here, `flups` overwrites the first user-provided information f_0 as represented in Figure 1. Also, we note that the half modes produced have an associated frequency of $\omega_k = \frac{2\pi}{2L}\left(k + \frac{1}{2}\right) = \frac{2\pi}{4L}(2k+1)$. Therefore they correspond to the odd frequencies that would have been obtained using a DFT on a domain of size $4N$ applying all the symmetries explicitly.

4) Even-odd boundary condition: The final combination is the even-odd case which is obtained using the type-III DCT, defined as

$$\tilde{f}_k = f(x_0) + 2 \sum_{j=1}^{N-1} f(x_j) \cos\left[\frac{2\pi}{2N} j \left(k + \frac{1}{2}\right)\right] , \quad (6)$$

with $k \in [0 \, ; \, N-1]$ and where `flups` overwrites the last user-provided information as illustrated in Figure 1. Similarly to the odd-even case, we note that the corresponding frequencies are the odd frequencies that would have been obtained using a DFT on a domain of size $4N$ applying explicitly all the boundary conditions.

Figure 1: Examples of odd-odd (○), odd-even (◇), even-odd (□), and even-even (△) boundary conditions. The shaded area represents the symmetry imposed by the boundary conditions. Data located there are fictitious. Filled shapes represent the points given to `fftw`, while the empty shapes are the data assumed by the BCs.

C. Semi-unbounded and unbounded boundary conditions

1) Unbounded directions: The unbounded boundary condition is imposed using the domain doubling technique of Hockney and Eastwood [15]. The algorithm extends the right-hand side to a domain of size $2L$ and fills the extension with zeros. The Green's function is also extended and symmetrized around L. The spectral representation of both extended fields $\tilde{G}_{ext}$ and $\tilde{f}_{ext}$ is then obtained with a DFT on the extended domain, and the periodic convolution is performed as a multiplication in the spectral space. We refer the reader to [1] for more details on the unbounded boundary condition treatment and various expressions for the Green's function.

Compared to the already mentioned approach, the generalization to node-centered data layout only affects the padding sizes, and the expressions for the Green's function remain unchanged. As illustrated in Figure 2 the right-hand side is extended by $N-2$ points, all set to 0, and we perform

the DFT on a domain from 0 to $2L$. On the Green's function side, the symmetry happens around $j = N$, and the domain is extended with $N - 2$ information.

(a) unbounded   (b) even/odd - unbounded

Figure 2: Extension of the right-hand side (●) and of the Green function(●) for the unbounded and semi-unbounded boundary condition. The values of the extended fields are shaded. For the semi-unbounded case, the domain's extension can be done on both sides of the domain and is here represented on the right end of the domain.

2) Semi unbounded directions: `flups` also supports semi-unbounded boundary conditions as a combination of an unbounded boundary condition at one end of the domain and a symmetry condition at the other end. As explained in [1], it relies on the domain doubling technique to impose the infinite boundary condition and on DSTs and DCTs to prescribe the correct symmetry while reducing memory usage. In this configuration, the right-hand side is first extended and padded on the unbounded side, unlike the fully unbounded case where we always extend it from L to $2L$. Then, a DST or DCT is executed on the extended domain to impose the proper symmetry condition. The Green's function is evaluated on the extended domain, and a DCT transform imposes the proper symmetry conditions required by the domain doubling technique.

As highlighted in Section II-B `fftw` does not consider the same amount of data when computing a DST or a DCT. Specifically when considering an even-unbounded boundary condition the right-hand side undergoes a DST while the Green's function a DCT. Therefore the spectral information obtained as a result of `fftw` has different wave-numbers and different sizes due to the use of `fftw`'s convention. We solve this issue by allocating extra space for the DST and using a common indexing for both the right-hand side and the Green's function.

III. Implementation

The implementation challenges of unbounded Poisson solvers as proposed in `flups` are almost equivalent to the one of a parallel distributed FFT transform. However a few key differences exist and are worth noting as they have motivated the authors of `flups` toward their own implementation.

- ability to mix different types of FFTs together (DFTs, DCTs, and DSTs) to support the different boundary conditions;
- the transform should happen in-place;

- the order of the transforms should be determined from the boundary conditions and not be imposed by the framework, to reduce the memory footprint and increase the solver performance;
- the unbounded boundary conditions should be supported, *i.e.* the FFT sizes might vary to accommodate zero-padding;

The original `flups` library satisfies those requirements but suffers from a plain approach and is not generalizable to node-centered data layout. The main problem is that the latter leads to an odd number of data to be distributed on a usually even number of processes, which results in a load in-balance very poorly managed by the original implementation. To offer flexibility and performance to the user we have significantly improved the communication strategies for both the existing implementations (all-to-all and non-blocking), and we have introduced a third one based on `MPI_Datatype`.

In this section, we detail the generalization and improvement of the communication strategies overarching `flups`, equivalent to the distributed FFT algorithm. For a pencil-based decomposition, the commonly used approach computes the forward or backward transform successively in the 3 dimensions. For each dimension, we first decompose the domain into pencils and send the data over to their respective process. The data transfers are either performed from the domain decomposition chosen by the user to the first pencil decomposition or from one pencil decomposition to another. Then we re-order the data (this operation is also known as the *shuffle*) and compute the FFT using `fftw` on the resulting continuous array. Finally, we repeat the whole process for the next dimension. In the rest of this section, we assume that the reader is familiar with the implementation of pencil-based distributed FFT and we refer to [1] for further details.

A. Communication strategies

The communications widely dominate the time-to-solution in a distributed FFT solver as already reported in several implementations [1, 2, 4]. To achieve the highest level of performance, we propose three strategies based on different `MPI` functionalities: an implementation using the all-to-all function `MPI_Ialltoallv` (noted a2a), an implementation relying on *persistent* requests and manual (un)packing of the data (noted nb), and an implementation based on non-blocking send and receives where `MPI_Datatype` is used to avoid the manual packing/unpacking (noted isr). Similar to the original implementation, all the `MPI` calls are performed in sub-communicators to reduce the memory footprint at large scale [16].

1) Implementation using an all-to-all: The most simple approach is to rely on `MPI_Ialltoallv` to perform the communications. This approach is summarized in Figure 3 and detailed in Algorithm 1. Each rank has a list of other ranks to send data to. The intersection of the data from the origin rank in the previous pencil decomposition with the destination rank in the new pencil decomposition leads to the definition of a *block of memory* that has to be transferred, noted b. For each rank we can pre-compute the list of blocks to send, b_{send}, and the ones to receive,

b_recv. Finally each block b corresponds to a location in the communication buffers (send and receive) and the user-data, noted $\text{buf}_\text{send}[b]$, $\text{buf}_\text{recv}[b]$, and $\text{data}[b]$ respectively.

With those definitions, the `a2a` approach consists in the following steps:

1) pack the non-regular data into the contiguous communication buffer in `pack()`;
2) use `MPI_Ialltoallv` to send the needed data to each of the corresponding ranks;
3) reset the data field to 0 in `reset()`, which is required to properly account for the unbounded boundary conditions and the use of in-place transforms;
4) wait for the communication to complete;
5) shuffling the data to realign them in memory in `shuffle()` and unpacking of the communication buffer to the data in `unpack()`.

Figure 3: Implementation of the `a2a` version. Data are packed from the user-provided buffer (▬) to the send buffer (▬). The communication is performed using `MPI_Ialltoallv`. The data are then shuffled in the receive buffer (▬) and copied back to the user buffer.

Algorithm 1: All-to-all implementation

foreach $b \in b_\text{send}$ **do**
  | $\text{buf}_\text{send}[b] \leftarrow$ `pack` $(\text{data}[b])$
end

`MPI_Ialltoallv`$(\text{buf}_\text{send}, \text{buf}_\text{recv})$
`reset`(data)
`MPI_Wait`$()$

foreach $b \in b_\text{recv}$ **do**
  | `shuffle`$(\text{buf}_\text{recv}[b])$
  | $\text{data}[b] \leftarrow$ `unpack`$(\text{buf}_\text{recv}[b])$
end

Although this approach is straightforward to implement, it suffers from a main issue: it does not expose the parallelization structure of the algorithm to MPI. It results in an implicit communicator-wide synchronization inherent to the `MPI_Ialltoallv` call and there is only limited room for optimization such as overlapping the different tasks. Therefore we expect the performance to be mitigated and driven by the MPI implementation.

2) Implementation using non-blocking persistent requests: To avoid the implicit barrier from the collective call and to expose more of the parallel structure of the algorithm to

MPI, we have implemented a version based on persistent non-blocking `MPI_Send` and `MPI_Recv`, referred to as `nb`. Although similar to the approach proposed in [1], our implementation contains several improvements designed to increase the parallel performance.

As for `a2a` we still have to perform the same `pack()`, `unpack()`, and `shuffle()` tasks. However we can overlap the computations, *i.e.* packing and unpacking, resetting the data field to 0, and shuffling the data, with the communication itself. This implementation is detailed in Algorithm 2 and summarized in Figure 4. Here b is still used to refer to a block of memory, and b_send and b_recv corresponds to the list of blocks to be sent and to be received respectively. We use the notation b_unpack to designate the list of blocks to unpack, *i.e.* copied from the receive buffer to the user-data.

This approach relies on two phases: a first one pre-computes and stores different `MPI_Requests`, which are used in the second one to communicate when needed. During the initialization phase, each block to be sent or received is associated with one of these `MPI_Requests`: $rqst_\text{send}$ or $rqst_\text{recv}$ respectively. The communication phase is made of four distinct steps:

1) for the requests in $rqst_\text{send}$, we manually pack the data into the continuous communication buffer and then start the corresponding request. This is done in the function `SendRqst` which takes as an argument B, an arbitrary list of blocks to be prepared and sent. For the requests in $rqst_\text{recv}$, no particular operation is needed except activating the requests using `MPI_Start`;
2) reset the buffer if all the send requests have completed;
3) test the completion of some of the requests in $rqst_\text{recv}$ and shuffle the one that have just completed;
4) unpack the shuffled requests if the buffer has been reset already.

As described in Algorithm 2, the four tasks are organized as a `for` loop relying on two compile-time variables n_batch and $n_\text{max-pending}$ to control the granularity of the different steps. The first one, n_batch, controls the number of requests gathered inside a batch and therefore treated one after another. The second one, $n_\text{max-pending}$, limits the total number of uncompleted send requests. The function `SendRqst` will therefore activate the minimum number of requests to not overtake any of those two thresholds.

In summary, the `nb` approaches offer a control on the asynchronous granularity. First, it starts all the *receive* requests, $rqst_\text{recv}$, and a first batch of *send* requests. Then, as long as there are ongoing send or receive requests, or block to unpack, the following steps are performed:

- if some b_send have to be sent, we compute the number of blocks to send with respect to n_batch and $n_\text{max-pending}$ and we send them;
- if all the b_send has been treated and if the user-provided data has not been reset yet, we reset the data field;
- if some b_recv has not been received, we force progress by calling `MPI_Testsome`, and we shuffle receive requests already completed;
- if the data field has been reset and if some blocks have been shuffled and are ready to be unpacked, we unpack them.

Figure 4: Implementation of the `nb` version. Data are copied from the user-provided buffer (▬) to the send buffer (▬). The communication is performed using *persistent non-blocking* communications. The data are then shuffled in the receive buffer (▬) and copied back to the user buffer.

In practice, we have observed the best performance with $n_{batch} = 1$. We attribute this behavior to the size of our communications which exceeds the eager / rendez-vous threshold. The messages can therefore not be sent right away and require a hand-shake to happen beforehand. There is then no gain in grouping the send-requests together. On the other side, requesting progress more frequently on the receive requests via a call to `MPI_Testsome` improves the performance.

3) Implementation using datatypes: From the previous non-blocking implementation we have observed that a significant time is spent packing and unpacking the data. Also we need to allocate the communication buffer to send the data which increases the memory footprint. To tackle that issue we propose a third possible implementation where we take advantage of `MPI_Datatype` to bypass the send-buffer allocation and packing on the send side. However, we still expect the latter to lead to additional overhead from the MPI implementation. On the receive side we still manually unpack the receive buffer to avoid waiting for the completion of all the send operations before starting the receive ones. As the send requests are non-blocking, waiting for their completion could lead to an overflow on the network with too many send requests started and no receive requests ready. This choice further allows us to overlap the shuffling of the received data with the overall communication scheme.

Compared to the `nb` implementation, the approach remains unchanged as illustrated in Figure 5. There are still four main steps: (1) the treatment of the send request, (2) the reset of the user-data, (3) the completion of the receive request and the shuffle of the associated buffer, and finally (4) the unpacking of the receive buffer in the user-data. The implementation differs in the treatment of the send request in the `SendRqst` function and with the use of non-blocking send (`MPI_Isend`) and receive (`MPI_Irecv`) instead of persistent requests. We also note that in `SendRqst` the packing is not needed anymore, as detailed in Algorithm 2.

Algorithm 2: Non-blocking implementations

n_{batch}, $n_{max\text{-}pending}$ *// User defined variables*
`RecvRqst` (b_{recv})
`SendRqst` (b_{send})
while *(*$b_{send} \neq \emptyset$ **or** $b_{recv} \neq \emptyset$ **or** $b_{unpack} \neq \emptyset$*)* **do**
  if *(*$b_{send} \neq \emptyset$*)* **then**
    $n_{ongoing} \leftarrow$ `MPI_Testsome` $(rqst_{send})$
    $n_{to\text{-}send} = \min(n_{max\text{-}pending} - n_{ongoing}, \quad n_{batch})$
    `SendRqst` $(b_{send}[n_{to\text{-}send}])$
  end

  if *(*$b_{send} = \emptyset$ **and** data buffer is not reset*)* **then**
    `reset` (data)
  end

  if *(*$b_{recv} \neq \emptyset$ *)* **then**
    `MPI_Testsome` $(rqst_{recv})$
    foreach $b \in b_{recv}$ **do**
      `shuffle` $(buf_{recv}[b])$
    end
  end

  if *(* data buffer is reset **and** $b_{unpack} \neq \emptyset$*)* **then**
    foreach $b \in b_{unpack}$ **do**
      $data[b] \leftarrow$ `unpack` $(buf_{recv}[b])$
    end
  end
end

――――――――――――――――――――――――――
――――

―― *non-blocking persistent requests*
Function `SendRqst` (B):
  foreach $b \in B$ **do**
    $buf_{send}[b] \leftarrow$ `pack` $(data[b])$
    `MPI_Start` $(rqst_{send}[b])$
  end

Function `RecvRqst` (B):
  `MPI_Startall` $(rqst_{recv}[b \in B])$

――――――――――――――――――――――――――
――――

―― *non-blocking datatypes*
Function `SendRqst` (B):
  `MPI_Isend` $(rqst_{send}[b \in B])$

Function `RecvRqst` (B):
  `MPI_Irecv` $(rqst_{send}[b \in B])$

IV. VALIDATION

The parameter space used in `flups` is very large due to its flexibility as it now supports 2 data layouts, 1000 boundary conditions, 8 Green's functions, and 3 communication strategies. All the potential combinations (more than 48,000) have been validated thoroughly as part of our continuous integration framework. Here for the sake of clarity, we present the validation of 3 different cases using the node-centered data layout. They address the standard use of `flups` when solving the Poisson equation with either fully spectral, fully unbounded, or semi-unbounded boundary

Figure 5: Implementation of the `isr` version. Data are sent and directly copied from the user-provided buffer (▮) to the receive buffer (▮) thanks to *non-blocking* communication and `MPI_Datatype`. The data are then shuffled in the receive buffer and copied back to the user buffer.

conditions (BCs). All the possible Green's functions are tested. Since all the implementations produce the same results, we only present the order of convergence obtained with the `a2a` version of the framework.

As previously proposed in [1] the test case is the Poisson Equation (1) with various boundary conditions. The infinite norm of the error computed as

$$E_\infty = \sup_{x,y,z}\{|\phi(x,y,z) - \phi_{ref}(x,y,z)|\} \,, \qquad (7)$$

where ϕ_{ref} is an analytical solution constructed as a product of 1-D functions:

$$\phi_{ref}(x,y,z) = X(x)\,Y(y)\,Z(z) \,. \qquad (8)$$

The functions X, Y, Z are chosen to match each set of boundary conditions. For example, sine and cosine with the proper wavelength are used with symmetric and periodic BCs while unbounded and semi-unbounded BCs are validated using Gaussian functions. Given the analytical solution, we then compute the corresponding right-hand side as the Laplacian of the reference solution:

$$f(x,y,z) = \frac{d^2 X}{dx^2}\,Y(y)\,Z(z) + X(x)\,\frac{d^2 Y}{dy^2}\,Z(z) + X(x)\,Y(y)\,\frac{d^2 Z}{dz^2} \,, \qquad (9)$$

where the appropriate functions for each considered case is given in Appendix B.

In the following sections, the Poisson equation is solved on a cubic domain of spatial extent $[0,L]$ in all directions. We use the Green's function as defined in [1]. More specifically, the one described in [14] hereafter referenced as `CHAT2` has a spectral accuracy for periodic or symmetric BCs and is of second order with unbounded conditions. The regularized kernels [17] are implemented from the order $m = 2$ to the order $m = 10$ together with the spectral-like regularization [18]. They are all named as a combination of the prefix `HEJ` followed by the respective order (spectral is labeled as order 0). Finally the Lattice Green's function [19] has a convergence order of two and is labeled as `LGF2`.

A. Domain with symmetric and periodic BCs

The chosen boundary conditions for this case are an even-even symmetry condition in the *x*-direction, an odd-

even symmetry condition in the *y*-direction, and a periodic condition in the *z*-direction. The associated reference solution can be found in Appendix B-A. The convergence results are shown in Figure 6 and match the expected convergence order. As the present case does not involve unbounded conditions the singular Green's function `CHAT2` provides the exact solution regardless of the resolution.

Figure 6: Convergence with symmetric and periodic BCs `CHAT2` (━●━), `LGF2`(━●━), `HEJ2`(━●━), `HEJ4`(━●━), `HEJ6`(━●━), `HEJ8`(━●━), `HEJ10`(━●━)

B. Fully unbounded boundary conditions

The reference solution for the fully unbounded case is given in Appendix B-B. As depicted in Figure 7, we observe a convergence corresponding to the theoretical order with all the Green's function. We want to highlight that here we have added the spectrally truncated kernel `HEJ0` achieving a spectral-like convergence.

Figure 7: Convergence with fully unbounded BCs `CHAT2` (━●━), `LGF2`(━●━), `HEJ2`(━●━), `HEJ4`(━●━), `HEJ6`(━●━), `HEJ8`(━●━), `HEJ10`(━●━ , `HEJ0`(━●━)

C. Domain with two semi-infinite and one fully unbounded BCs

For the semi-unbounded BCs an even symmetry is imposed on the right side in the *x*-direction while an odd

symmetry is applied on the left side in the z-direction. All the remaining boundaries are unbounded. The reference solution is then computed as indicated in Appendix B-C. As shown in Figure 8 all the Green's function reach the expected convergence orders.

Figure 8: Convergence with semi unbounded BCs CHAT2 (●), LGF2(●), HEJ2(●), HEJ4(●), HEJ6(●), HEJ8(●), HEJ10(●), HEJ0(●)

V. APPLICATION: THE BIOT-SAVART SOLVER

To demonstrate the flexibility of flups and its use in practice we now consider the Bio-Savart equation, a variation on the standard Poisson equation, given by

$$\nabla^2 u = \nabla \times f \ . \tag{10}$$

This relation is particularly useful in application such as computational fluid dynamics, when one needs to recover the velocity from the vorticity field. While the equation has its own Green's functions [17], one could take another approach that extends the work done previously and relies on the flexibility of flups. First compute the forward FFT of the rhs $\hat{f}$, then compute the curl in the spectral space, compute the convolution with $\hat{G}$, and finally compute the FFT backward.

To evaluate the curl in spectral space we have to properly apply a spectral derivative on the result of the forward FFT. With periodic or unbounded boundary conditions the result of the forward FFT is complex, and therefore the evaluation of a derivative becomes

$$\frac{\partial}{\partial x} f \quad \rightarrow \quad (i\omega_k)\ \tilde{f}\ . \tag{11}$$

where the differential operator $(i\omega_k)$ is a purely imaginary number, and the derivation is then spectrally accurate. Depending on the targeted application it can also be useful to consider a finite difference approximation instead of the actual derivative. Then the following expressions are used for the order 2:

$$\frac{\partial}{\partial x} f + \mathcal{O}\left(h^2\right) \quad \rightarrow \quad \frac{i}{h} \sin\left(\omega_k\, h\right) \tilde{f}\ , \tag{12}$$

the order 4:

$$\frac{\partial}{\partial x} f + \mathcal{O}\left(h^4\right) \quad \rightarrow \quad \frac{i}{h} \left(\frac{4}{3} \sin\left(\omega_k\, h\right) - \frac{1}{6} \sin\left(\omega_k\, 2h\right)\right) \tilde{f}\ , \tag{13}$$

and the order 6:

$$\begin{aligned} &\frac{\partial}{\partial x} f + \mathcal{O}\left(h^6\right) \\ &\rightarrow \quad \frac{i}{h} \left(\frac{3}{2} \sin\left(\omega_k\, h\right) - \frac{3}{10} \sin\left(\omega_k\, 2h\right) + \frac{1}{30} \sin\left(\omega_k\, 3h\right)\right) \tilde{f}\ . \end{aligned} \tag{14}$$

For the even or odd BCs, applying a derivative will inverse the original condition: an even condition will become odd, and an odd one will become even. Therefore, the type of the backward FFT used must be adapted in order to reflect that change. However, changing a DST for a DCT (and the opposite) should be done with care, as already discussed earlier. If the input of a DCT is f then the output $\tilde{f}$ corresponds to the complex number $(\tilde{f} + 0i)$. On the other hand, if the input of a DST is f, then its output $\tilde{f}$ corresponds to the complex number $(0 - i\tilde{f})$. Consequently, if a DST is used as the forward transform, taking the derivative of the output leads to $(i\omega_k)(0 - i\tilde{f}) = \tilde{f}\omega_k$ which can be used directly as the input of the backward DCT. On the other hand, if a DCT is used as the forward transform, the output is $(\tilde{f} + 0i)$ and the first derivative is $(i\omega_k)\tilde{f} = i\omega_k\tilde{f}$ whose sign must be changed to be used as the input of the DST leading to $-\tilde{f}\omega_k$. The same approach applies to the derivatives computed using the finite differences, where ω_k is replaced by the appropriate formula.

The obtained Biot-Savart solver is validated with the case of a vortex tube aligned in the z-direction. The tube is centered within a cubic domain of size $[0, L]^3$ and fully unbounded boundary conditions are imposed in the x- and y-direction while symmetry conditions are used in the z-axis: ω_x and ω_y undergo odd symmetry conditions while ω_z satisfies an even symmetry on the domain boundaries.

The tube is compact and has the following expression:

$$\omega(x, y, z) = \left\{0, 0, -\omega_z(r)\right\}\ , \tag{15}$$

where r is defined as $r = \sqrt{(x - 0.5L)^2 + (y - 0.5L)^2}$ and $\omega_z(r)$ is computed as

$$\omega_z(r) = \begin{cases} \frac{1}{2\pi} \frac{2}{R^2} \frac{1}{E_2(1)} \exp\left(-\frac{1}{\left(1 - \left(\frac{r}{R}\right)^2\right)}\right) & \text{if } r <= R \\ 0 & \text{otherwise,} \end{cases} \tag{16}$$

using R as the radius of the vortex tube, and E_2 is the generalized exponential integral function.

The corresponding analytical velocity can be retrieved through [20]:

$$\mathbf{u}(x, y, z) = \left\{-\sin(\theta)\, u_\theta(r), \cos(\theta)\, u_\theta(r), 0\right\}\ , \tag{17}$$

where $u_\theta(r)$ is given by

$$u_\theta(r) = \begin{cases} \frac{1}{2\pi r} \left[1 - \frac{1}{E_2(1)} \left(1 - \left(\frac{r}{R}\right)^2\right) E_2\left(\frac{1}{1 - \left(\frac{r}{R}\right)^2}\right)\right] & \text{if } r <= R \\ \frac{1}{2\pi r} & \text{otherwise.} \end{cases} \tag{18}$$

Figure 9 shows the convergence of the error computed as the infinite norm of the velocity field when using the spectral differentiation of the curl[1]. One can also replace the differentiation by an approximation using finite differences

---

[1]We have corrected a typo from [1, B.10], which should be $\frac{1}{8\pi}\left(\pi - 6 + 2\log\left(\frac{\pi}{2} r_{eq}^2\right)\right)$ when $k_z = 0$.

of order 6, whose convergence is given in Figure 10. For further details, the convergence of the finite difference of order 2 and 4 are given in Section C (Figure A and Figure B respectively). As expected the measured convergence orders correspond to the minimum between the differentiation order and the kernel order.

Figure 9: Convergence of the Biot-Savart solver using spectral differentiation CHAT2 (—•—), HEJ2(—•—), HEJ4(—•—), HEJ6(—•—), HEJ8(—•—), HEJ10(—•—), HEJ0(—•—)

We also note that HEJ0 kernel is the only one not achieving the expected convergence. We attribute this behavior to the truncated infinite sum taking place when computing this non-singular Green's function [18] in the case of only two unbounded directions. The truncation entails an approximation of the kernel and affects its accuracy. In this case, the kernel is consequently bounded to the second order.

Figure 10: Convergence of the Biot-Savart solver using 6th order differentiation CHAT2 (—•—), HEJ2(—•—), HEJ4(—•—), HEJ6(—•—), HEJ8(—•—), HEJ10(—•—), HEJ0(—•—)

VI. PARALLEL PERFORMANCE ANALYSIS

This section presents the results of performance tests carried out on different massively parallel architectures. To obtain the results presented in this section we have set n_{batch} to 1 and $n_{max\text{-}pending}$ to INT_MAX for both the isr and the nb implementations. From a technical perspective, flups has been compiled with the -DNDEBUG -O3 flags. We have used mpich 4.1a1 compiled with -enable-fast=O3,ndebug,alwaysinline as the MPI implementation. On Infiniband (IB)-based networks the communication library is ucx 1.13.1, while libfabric 15.0.0 from the vendor has been used for the slingshot-based infrastructure. Still for IB networks the simulations were run with the DC transport layer.

First, we compare our nb and isr implementation against the accFFT library. Then we study our scalability for the three proposed implementations. Finally, we provide a comparison of the performance on three different systems.

A. Comparison with accFFT

They are many other implementations of the distributed FFT algorithm, but only a handful of them provide the *real-to-complex* FFT computation. Among them, accFFT [2], an implementation for both CPU and GPU partitions, is usually considered as one of the fastest [3].

To fit in the framework proposed by accFFT we had to use a very specific test-case, here again highlighting the flexibility of flups. We use a cell-centered data-layout and perform a forward 3D FFT followed by a backward 3D FFT. The first topology is aligned in the direction of the first FFT so that only two topology switches are performed and three 1D FFTs. In our case we then start with pencils in the *X* direction while accFFT starts with pencils in the *Z* direction by default. The number of unknowns per rank is fixed to 256^3, and the process distribution over the domain is given in Section E-A. To have the fairest comparison possible, a single executable calling both libraries is created, with both libraries compiled using the same binaries of FFTW and MPI. Both libraries call FFTW with the FFTW_MEASURE flag so that the differences in timings are only due to differences in the implementation. The tests have been done with exclusive node allocations on MeluXina, detailed in Table D.

The obtained time-to-solution and weak efficiency for accFFT and the isr and nb implementation is presented in Figure 11 using up to 16,384 cores (128 nodes). For reference, we also convert those times-to-solution into throughput per rank [MB/sec], provided in Table II. This metric is particularly useful to compare time-to-solution results across infrastructure and testcases. In Figure 11 we first note that the isr implementation and the accFFT one are very similar for a small number of nodes. For a larger count, the isr implementation is slightly slower yet remains very competitive compared to accFFT. The nb implementation follows another path where the penalty of manual packing and unpacking vanishes as the node count increases. From Table II, at 128 nodes the nb implementation runs 27.7% faster than accFFT. We attribute this difference to the overhead coming from the use of MPI_Datatype, which can also be observed in the scalability of the nb and isr approaches presented in Section VI-C. We conclude from this comparison that flups offers more flexibility compared to accFFT, but it does not come with significant performance degradations.

Figure 11: Comparison with `accFFT`: weak scaling time-to-solution for `accFFT` (▮), the `isr` (▮), and the `nb` (▮) versions of `flups`. A forward and a backward 3D FFTs are performed with 256^3 per rank on MeluXina. The hashed part of the bar corresponds to the forward transform, the plain part represents the backward transform.

N nodes	1	2	8	32	128
`accFFT`	36.23	33.10	30.57	25.25	18.51
`isr`	36.13	34.02	28.80	23.34	18.62
`nb`	27.80	27.09	25.98	26.34	25.62

Table II: Comparison with `accFFT`: throughput per rank [MB/sec] for a forward and backward FFT.

B. General comments on weak and strong scalability

Ahead of our weak and strong scalability analysis, we would like to refer the reader to Section D for details on the performance metrics used in this section. In particular we will use the sequential percentage of a program, β, as a measure of the quality of our implementation, together with the speedup s_P (strong scalability) and the efficiency η_P (weak scalability).

In this section, we present both weak and strong scalability tests. The tests were performed on the CPU nodes of MeluXina (LuxConnect's Data Center DC2, Luxembourg). The CPU partition is made of 573 nodes, each of them composed of 2 AMD EPYC 7H12 CPUs with 64 cores per CPU, which make a total of 128 cores per node. The nodes are connected with InfiniBand HDR200 Gb/s and organized in a Dragonfly+ topology. In both cases, we considered a 3D gaussian function in a fully unbounded domain (see Section IV-B for details on the test cases). All the times-to-solution are presented as the average execution times over the ranks.

C. Weak Scalability

We start our weak scalability test on a single node and run it up to 384 nodes, *i.e.* $49,152$ cores, where we have used 96^3 points per core in the user domain with the process distribution given in Section E-B.

a) Time-to-solution: Figure 12 presents the evolution of the time-to-solution needed for a solve for each com-

munication strategy, while Figure 13 shows the associated weak efficiency. On a single node, the `nb` strategy is close to the `a2a` one, yet slightly faster. The timing difference between both implementations remains constant when increasing the number of nodes, achieving therefore a similar weak efficiency. On the other side, the `isr` method is the fastest one up to 16 nodes. Afterwards, the communication timings reach those of the `a2a` approach. The difference in scalability between the implementation relying on the `MPI_Datatype` (`isr`) and those using manual packing and unpacking (`a2a` and `nb`) is significant. Even if the time-to-solution is lower at small count, the gain of the `MPI_Datatype` appears to vanish when increasing the partition count. This behavior varies with the different MPI versions as the treatment of `MPI_Datatypes` is specific to each implementation. Finally, we attribute the peaks happening on large partitions to the congestion on the network at the time of the testing (Dragonfly topology).

To have a meaningful comparison with a periodic case such as the one used in the previous section, the throughput per rank obtained for a fully unbounded domain in Table III must be normalized by a factor of $14/3$. This factor comes from the doubling technique in which the first transform is performed on $2N$ data, the second on $4N$, and the last on $8N$, which makes an average of $14/3N$ data per transform for the whole 3D FFT. Compared to the results presented in Table II we here apply three topology switches instead of 2.

Figure 12: Weak scaling: time-to-solution for `a2a`(▮), `isr`(▮), and `nb`(▮) versions. Tests on MeluXina with 96^3 unknowns per rank and a fully unbounded testcase.

b) Step-by-step analysis: For further details we distinguish the different steps from within a call to `flups` inside the bar plots, see fig. 12. The crossed section represents the time spent performing computations only: 1D FFTs, spectral multiplication, and copy of the data provided by the user to the work buffer. The lined section shows the time spent in computations overlapping communication: copying back and forth data from the work buffer to the communication buffers, shuffling the data, and resetting the work buffer to 0. The remaining non-hatched regions correspond to the time spent only communicating. The different colors shades further differentiate the topology switches, the darker color corresponds to the first topology switch, and the lighter color to the last one.

Figure 13: Weak scaling: efficiency $\eta_{P,w}$ for a2a(—●—), isr(—●—), and nb(—●—) versions. Tests on MeluXina with 96^3 unknowns per rank and a fully unbounded testcase.

N nodes	1	2	8	64	128
a2a	30.34	29.85	27.21	25.94	24.33
nb	32.14	31.54	29.59	29.19	26.57
isr	40.94	38.39	32.49	26.92	23.82

Table III: Weak scaling: throughput per rank [MB/sec] for a solve with the three different code versions. To account for the domain-doubling technique for unbounded BCs a normalization factor of 14/3 has been applied.

problem size is fixed to 1280^3 unknowns and the process distribution is given in Section E-C.

Figure 14 shows the strong scalability time-to-solution. As in the weak scalability, the isr approach has the shortest resolution timings for a small number of MPI ranks. It is followed by the nb and finally the a2a version. The timings gap between the isr and the other methods steadily decreased when increasing the nodes number. In agreement with those results, the computation-only part of the code shows a linear speed-up: the computation time is inversely proportional to the number of resources.

Figure 14: Strong scaling: time-to-solution for a2a(▮), isr(▮), and nb(▮) versions. Tests on MeluXina with 1280^3 unknowns in total on a fully unbounded testcase.

In Figure 15 we estimate the percentage of the software running in parallel, β, now based on the effective speed-up s_P. The values found using the speedup are similar to those of the weak scaling tests, stating that approximately $99.5-99.8\%$ of our implementation is parallelized, while the remaining is sequential. We also observe that the scalability gap between the implementations illustrates the associated software latency: the nb approach is expected to involve more software operations than the a2a, and the isr has a higher latency than the nb due to the allocation of extra buffers in the MPI implementation to pack and unpack the data.

As expected in the case of a weak scalability the computation-only part of the code scales perfectly and represents a fixed cost regardless of the communication strategy. The isr timings for the computation (hatched colored region, overlapping with communication in the case of isr and nb) are the fastest, as the isr implementation removes the manual packing. However, the benefits are lessened as the time spent in the communication-only part of the algorithm increases. As explained in Section III, the a2a strategy only resets the user buffer to zero while waiting for the all-to-all communication to complete and the other operations are done sequentially. This translates into timings associated with the computation-communication overlap section of the code slightly behind the timings of the other strategies. Finally, in a fully unbounded case, the domain is expanded between the topology switches. It increases the number of points to be exchanged between the MPI processes and raises the communication cost of the topology switches. Also, due to rank distribution among the nodes, the first topology switch is mostly happening intra-node, while the second and third ones are inter-node mostly. Those two factors together explain the increasing time from one topology switch to another.

c) Weak efficiency: The associated weak efficiency of the software is presented in Figure 13. In our cases, increasing the number of resources leads to an increase of the number of communications and congestion on the network. As expected from the previous results, the isr approach shows the poorest scalability, and we can estimate its theoretical sequential percentage to $\beta = 0.5\%$. We also note that the nb and the a2a version have very similar weak scaling efficiency. Their serial percentage is estimated at $\beta_{nb} \approx 0.2\%$. At the light of those results we anticipate the nb version, having both smaller timings and higher efficiency, to be the version the most suited for very large-scale simulations.

D. Strong scalability

Similarly to the weak scalability testing, the strong analysis covers a range from one to 384 nodes, where the total

E. Comparison of main European systems

We present in this section the results of the same weak scalability test on three main European systems: Lumi, Vega, and MeluXina, summarized in Table D. Vega (Slovenia) is equipped with similar nodes as MeluXina (AMD 7H12, 128

Figure 15: Strong scaling: speedup s_P for `a2a`(—●—), `isr`(—●—), and `nb`(—●—) versions. Tests on MeluXina with 1280^3 unknowns in total on a fully unbounded testcase.

Figure 16: Implementation `nb`: time-to-solution on Vega (▮▪), MeluXina (▮▪), and Lumi (▮▪). Weak scalability tests performed with 96^3 unknowns per rank on a fully unbounded test case.

Figure 17: Implementation `nb`: Weak efficiency $\eta_{P,w}$ on Vega (—●—), MeluXina (—●—), and Lumi (—●—). Weak scalability tests performed with 96^3 unknowns per rank on a fully unbounded test case.

cores/node) but with slower interconnect: IB-HDR 100Gb/s instead of 200Gb/s for MeluXina . Lumi (Finland) and specifically on the Lumi-C partition each node has two AMD EPYC 7763 CPUs with a total of 128 cores per node, which are connected with a 200 Gb/s slingshot-11 network. Vega has then the slowest bandwidth and Lumi CPUs have a slightly faster CPUs clock speed.

Figure 16 and Figure 17 show the results of the weak scalability tests for the `nb` approach. For the interested reader, the throughput per rank as well as the results for the `isr` and `a2a` versions are presented in Section F. On a few nodes, the time-to-solutions are almost similar for all the architectures. However, the timings diverge with the increasing number of nodes. The Vega timings increase steeply than for Lumi and MeluXina. Lumi results are close to Meluxina ones, and the latter shows the best weak efficiency with a time increase of only 20% when multiplying the resources by 128. As discussed in Section VI-C, a weak scalability test raises the number of resources together with the number of unknowns hence increasing the number of communications and the network congestion. The Vega result is therefore explained by a lower bandwidth which saturates with the growing resources. On the other hand, we attribute the smaller times of MeluXina compared to Lumi to the hardware differences, the first being equipped with IB interconnect while the second uses slingshot technology. As in Section VI-C, we note that the computation time remains constant regardless of the number of nodes. Lumi spends slightly less time on computations while MeluXina and Vega have precisely the same computations timings. We attribute those differences to the CPUs properties as Vega and MeluXina have identical CPUs whereas Lumi has a slightly higher clock speed.

VII. CONCLUSIONS

Massively distributed FFT transforms have numerous applications and in particular in the PDE resolution realm. However, the contributions usually proposed in the computer science field fail to address important requirements to provide the user with a flexible, yet performant and scalable library. Relying on our expertise in computational fluid dynamics, we propose improvements to the `flups` software [1] to bridge this gap: the treatment of both node-centered and cell-centered data layouts as well as faster communication strategies exploring several possible implementations. The resulting interface is built such that the user only provides the number of points in the 3D Cartesian grid, the desired boundary conditions (even, odd, periodic or unbounded) and the library automatically orders the sequence of FFTs, extends the domain to handle unbounded directions, and performs the forward and backward transforms.

At a methodological level, we first modify the numbering conventions to accommodate the different types of FFTs provided by `fftw` and required for the cell- and node-centered data layouts. Then, we present three implementation strategies for a distributed FFT. First, the well-known `a2a` implementation relies on the commonly

used `MPI_Ialltoallv` function which implies a very strong synchronization and exposes very few parallelizations to the MPI library. Then, the `nb` approach, which relies on non-blocking persistent requests and manual packing/unpacking, exploits the possibility of a very fine-grained parallelization. This method makes the synchronization more explicit, through the parameters n_{batch} and $n_{max\text{-}pending}$ and hence reduces the overhead of the implementation. Finally, we explore the use of `MPI_Datatypes` to reduce the memory footprint of the solver, an approach named `isr`. This optimization removes the need for manual packing and is implemented through non-blocking send and receives. Both the `nb` and the `isr` implementations share a very similar structure which allows us to attribute the difference in performance to the use of `MPI_Datatypes`, a currently active subject for the different MPI implementations. To prove the flexibility of the proposed library, we demonstrate the use of `flups` to solve the Biot-Savart equation in Section V. This requires a special operation in spectral space, as well as different real-to-real FFTs in the backward and the forward transform. To our knowledge, no other library offers this level of convenience for the user.

The non-blocking approaches (`isr` and `nb`) are first compared to the `accFFT` library[2] in Section VI-A. The `nb` strategy demonstrates 27% faster time-to-solution over a large range of core count, while the `isr` implementation is as fast as `accFFT`. We conclude that our implementation is as fast, if not faster than one of the fastest implementation of the distributed FFT on CPU. Moreover, the flexibility introduced for the user does not reduce the achieved performance. Then in Section VI-C and in Section VI-D we focus on the scalability of our implementation both from a weak and a strong perspective. The `a2a` is observed to be significantly slower yet to achieve a good scalability, which is expected due to the implicit barrier on the sub-communicator. Regarding the non-blocking implementation, the `isr` is the fastest on a small count of nodes, but the advantage vanishes over a larger partition as the `nb` approach has a better scalability. Both the `nb` and the `a2a` implementation achieve an impressive weak and strong scalability with a sequential part of the implementation β below 0.5%. The strong scalability results follow the trends observed in the weak scalability, with slightly better estimates for β. Finally we apply our test case to three different leading European clusters: Lumi, Vega, and MeluXina in Section VI-E. The time-to-solution are compared between the clusters and as expected the better bandwidth available on MeluXina provides a faster time-to-solution on large partitions.

With the proposed improvements and changes, `flups` is now a highly flexible and performant distributed FFT framework, tailor-made for scientists and in particular computational fluid dynamics applications. We ambition here to bridge the gap between the numerous contributions in the computer science field, focusing mainly on performance, and the actual need of the user, which is a highly efficient and versatile framework to be used with lots of different configurations. Our contribution in this work also aims to provide a reference in terms of performance metrics with scalability tests on large partitions and on different architectures.

In the future we will further develop `flups` to exploit heterogeneous architectures with a particular focus on the `MPI+X` approach as proposed in the latest `MPI` standard. The newest additions indeed provide opportunities to reduce the rank count, one identified bottleneck with large partitions, as well as to exploit the concept of *streams* with threads and GPUs. This future direction aims at addressing the missing GPU implementation of this work.

REFERENCES

[1] D.-G. Caprace, T. Gillis, and P. Chatelain, "Flups: A fourier-based library of unbounded poisson solvers," *SIAM Journal on Scientific Computing*, vol. 43, no. 1, pp. C31–C60, January 2021. [Online]. Available: https://doi.org/10.1137/19M1303848

[2] A. Gholami, J. Hill, D. Malhotra, and G. Biros, "Accfft: A library for distributed-memory FFT on CPU and GPU architectures," *CoRR*, vol. abs/1506.07933, 2015. [Online]. Available: http://arxiv.org/abs/1506.07933

[3] A. Ayala, S. Tomov, P. Luszczek, S. Cayrols, G. Ragghianti, and J. Dongarra, "Fft benchmark performance experiments on systems targeting exascale," Tech. Rep. ICL-UT-22-02, 2022-03 2022.

[4] A. Ayala, S. Tomov, A. Haidar, and J. Dongarra, *heFFTe: Highly Efficient FFT for Exascale*, 06 2020, pp. 262–275.

[5] L. Dalcin, M. Mortensen, and D. E. Keyes, "Fast parallel multidimensional fft using advanced mpi," *Journal of Parallel and Distributed Computing*, vol. 128, pp. 137 – 150, 2019. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S074373151830306X

[6] D. Pekurovsky, "P3dfft: A framework for parallel computations of fourier transforms in three dimensions," *SIAM Journal on Scientific Computing*, vol. 34, no. 4, pp. C.192–C.209, 2012, copyright - © 2012, Society for Industrial and Applied Mathematics; Dernière mise à jour - 2012-09-17. [Online]. Available: https://search-proquest-com.proxy.bib.ucl.ac.be:2443/docview/1033558975?accountid=12156

[7] A. Pope, D. Daniel, and N. Frontiere, "A stand-alone version of hacc's distributed-memory, pencil-decomposed, parallel 3d fft." 2017.

[8] D.-G. Caprace, P. Chatelain, and G. Winckelmans, "Wakes of rotorcraft in advancing flight: A large eddy simulation study," *Physics of Fluids*, vol. 32, no. 8, p. 087107, 2020.

[9] T. Gillis, Y. Marichal, G. Winckelmans, and P. Chatelain, "A 2d immersed interface vortex particle-mesh method," *Journal of Computational Physics*, vol. 394, pp. 700–718, 2019. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0021999119303717

[10] J. Gabbard, T. Gillis, P. Chatelain, and W. M. van Rees, "An immersed interface method for the 2d vorticity-velocity navier-stokes equations with multiple bodies," *Journal of Computational Physics*, vol. 464, p. 111339, 2022.

[11] M. Frigo and S. G. Johnson, "The Design and Implementation of FFTW3," *Proceedings of the IEEE*, vol. 93, no. 2, pp. 216–231, 2005,

special issue on Program Generation, Optimization, and Platform Adaptation.

[12] A. Ayala, S. Tomov, P. Luszczek, S. Cayrols, G. Ragghianti, and J. Dongarra, "Interim report on benchmarking fft libraries on high performance systems," University of Tennessee, Tech. Rep., 2021.

[13] K. Czechowski, C. Battaglino, C. McClanahan, K. Iyer, P.-K. Yeung, and R. Vuduc, "On the communication complexity of 3d ffts and its implications for exascale," in *Proceedings of the 26th ACM International Conference on Supercomputing*, ser. ICS '12. New York, NY, USA: Association for Computing Machinery, 2012, pp. 205–214. [Online]. Available: https://doi.org/10.1145/2304576.2304604

[14] P. Chatelain and P. Koumoutsakos, "A Fourier-based elliptic solver for vortical flows with periodic and unbounded directions," *Journal of Computational Physics*, vol. 229, no. 7, pp. 2425–2431, 4 2010.

[15] R. Hockney and J. Eastwood, *Computer Simulation using Particles*. Taylor & Francis, Inc. Bristol, PA, USA, 1988.

[16] *Memory Compression Techniques for Network Address Management in MPI*, 2017.

[17] M. Hejlesen, J. Rasmussen, P. Chatelain, and J. Walther, "A high order solver for the unbounded Poisson equation," *Journal of Computational Physics*, vol. 252, pp. 458–467, 2013.

[18] M. M. Hejlesen, G. Winckelmans, and J. H. Walther, "Non-singular green's functions for the unbounded poisson equation in one, two and three dimensions," *Applied Mathematics Letters*, vol. 89, pp. 28–34, 2019. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0893965918303264

[19] P.-G. Martinsson and G. J. Rodin, "Asymptotic expansion of lattice Green's function," *Proceedings: Mathematical, Physical and Engineering Sciences*, vol. 458, no. 2027, pp. 2609–2622, 2002.

[20] G. S. Winckelmans, "Vortex Methods," in *Encyclopedia of Computational Mechanics*, 2nd ed. John Wiley & Sons, Ltd, 2004, vol. 3, pp. 129–153.

[21] S. Chunduri, T. Groves, P. Mendygral, B. Austin, J. Balma, K. Kandalla, K. Kumaran, G. Lockwood, S. Parker, S. Warren, N. Wichmann, and N. Wright, "Gpcnet: Designing a benchmark suite for inducing and measuring contention in hpc networks," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '19. New York, NY, USA: Association for Computing Machinery, 2019. [Online]. Available: https://doi.org/10.1145/3295500.3356215

[22] G. M. Amdahl, "Validity of the single processor approach to achieving large scale computing capabilities," in *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*, ser. AFIPS '67 (Spring). New York, NY, USA: Association for Computing Machinery, 1967, pp. 483–485. [Online]. Available: https://doi.org/10.1145/1465482.1465560

[23] J. L. Gustafson, "Reevaluating amdahl's law," *Commun. ACM*, vol. 31, no. 5, pp. 532–533, may 1988. [Online]. Available: https://doi.org/10.1145/42411.42415

Pierre Balty obtained his Master degree from UCLouvain in 2019 and is currently a PhD student and teaching assistant in the institute for Mechanics, Material, and Civil engineering at UCLouvain. His research focuses on Lagrangian numerical methods, their deployment on distributed systems, and their applications to wind energy.

Philippe Chatelain obtained his Ph.D. from Caltech in 2005 and held a research associate position at ETH Zurich till 2009. He is currently full Professor at UCLouvain, leading the Vortex and Turbulence research group. His research interests cover fluid mechanics, Lagrangian numerical methods, their deployment in HPC environment, and their application to fundamental and applied problems in bio-propulsion, aeronautics, and wind energy.

Thomas Gillis obtained his Ph.D. from UCLouvain in 2019. At the time of this work he was postdoctoral researcher at UCLouvain and visiting scientists at MIT. His research focuses on distributed systems, lossless compression for PDEs, and scalable and performant framework for computational fluid dynamics. Thomas is now part of the Argonne National Lab working on distributed systems.

APPENDIX A
IMPLEMENTATION: PERFORMANCE STRATEGIES

Throughout the development of the library we have established different strategies to improve the performance. The first one is to proceed first with the *intra-node* communications with a special communicator before the *inter-node* ones. As this optimization is usually also done in the MPI implementation, we have not measured a significant difference in terms of time-to-solution. In this section we describe two other strategies we have used. First a new way of distributing the data through the MPI ranks to avoid imbalance between the nodes and a specific order to proceed to the inter-node communications.

A. Load balancing - distribution of unknowns

To simplify the notation in this section we note the integer division by /. Different approaches exist to distributed N unknowns on P ranks:

- ranks from 0 to $N\%P$ have $N/P + 1$ data and ranks $> N\%P$ have N/P data as originally implemented in `flups`;
- a rank r gets its first data index from $(r\,N)/P$.

In `flups`, the computation of the communications requires such a formula to be invertible, *i.e.* we have to compute the corresponding rank for a data index and the data that is attributed to a specific rank index. While the first approach can be easily inverted (getting the rank index r for a specific data), the unknowns are poorly distributed over the ranks as the first $N\%P$ ranks will get more data. When considering multiple ranks per node it results in a subset of nodes having more data than the others and therefore in a significant imbalance between nodes.

This issue is solved with the second approach which spreads the excess data over the whole rank range. However, getting the rank from a data index has been impossible for us, or at least in an efficient way. To combine the benefit of the two approaches, we propose another distribution that both distributes the excess data over the whole rank range and can be easily inverted.

As with the other methods, all ranks will get at least $B = N/P$ unknowns, also referred to as the baseline. To distribute the remaining $R = N\%P$ ones, we create R groups of ranks. Each group has $S = P/R$ ranks (where S stands for stride), except the last one which might be a special case. The last rank of each group will get 1 excess data, then each group has a total of $(S\,B + 1)$ data. Here again the last group might be special and not get any +1 as highlighted in the example below. The first index for data attributed to a rank r is then obtained as

$$r\left(\frac{N}{P}\right) + \min\left\{\frac{r}{S}\,;\,R\right\}\,. \tag{a}$$

To inverse the relation and obtain the rank corresponding to a given data index i, we first identify the group index and then add the rank index within the group:

1) the group index where the data i is located is given by

$$g_i = \min\{i/(S\,B + 1)\,,\,R\}\,, \tag{b}$$

where the min ensures that edge cases do not over-estimate the group id;
2) within the group, the local data index is now given by $i_L = (i - g_i\,(S\,B + 1))$;
3) then the local rank attributed to the local index i_L is $r_L = i_L/B$, where we have to bound r_L to $S - 1$ to if $g_i < R$ with $r_L = \min\{i_L/B, S - 1\}$.

The rank attributed to the data index i is finally obtained as $r = g_i\,S + r_L$.

To illustrate our approach we consider an edge case with $N = 32$, $P = 7$. Then following the formula we obtain $B = 4$, $R = 32\%4 = 4$, and $S = 1$. The distribution is then given by $[5\,,\,5\,,\,5\,,\,5\,,\,4\,,\,4\,,\,4]$. If we want to get the rank for the data id of $i = 14$, then we obtain $g_i = 14/5 = 2$, $i_L = 4$, and $r_L = \min\{1,0\} = 0$. The final rank is then $r = 2$. Similarly if we want to get the rank for the data id $i = 27$, then $g_i = 27/5 = \min\{5\,,\,4\} = 4$, $i_L = 7$, and $r_L = 1$. The final rank is then $r = 4 + 1 = 5$. We note here that with this specific example the distribution is not better than the one already used in `flups`. However for more regular configurations we obtain a more homogeneous distribution (e.g. $N = 32$, $P = 6$).

1) Order of the send requests based on the destination rank: As the communication is done in an all-to-all manner, each rank in a sub-communicator interacts with all the other ranks belonging to the same sub-communicator. An intuitive way to implement such a scheme is to have all the ranks start their request following the rank indexing of the sub-communicator. For example, all the send requests from all the ranks inside the sub-communicator will first be addressed to rank 0, then to rank 1, and so on. As stated in [21], this may lead to endpoint congestion and lower the code performance. To reduce the network congestion, the ranks communicate with others in ascending order, starting with their neighbor in the indexes list. In that case, the rank r first sends its request to the rank having the index $r + 1$, then to the rank indexed $r + 2$, etc. The repartition of the active send requests across the different receivers is hence improved and it reduces the communication overheads.

APPENDIX B
ANALYTICAL EXPRESSIONS USED FOR THE VALIDATION

This section contains the details analytical expressions used for the validation of `flups` in Section IV. The Poisson equation is solved on a cubic domain of spatial extent $[0, L]$ in all directions.

A. Domain with symmetric and periodic BCs.

$$\phi_{ref}(x, y, z) = \cos\left(\pi\frac{x}{L}\right)\sin\left(\frac{5\pi}{2}\frac{y}{L}\right)\sin\left(8\pi\frac{z}{L}\right)\,. \tag{c}$$

B. Fully unbounded boundary conditions

$$\phi_{ref}(x, y, z) =$$
$$\exp\left(10\left(3 - \frac{1}{1 - \left(\frac{2x}{L} - 1\right)^2} - \frac{1}{1 - \left(\frac{2y}{L} - 1\right)^2} - \frac{1}{1 - \left(\frac{2z}{L} - 1\right)^2}\right)\right)\,.$$
$$\tag{d}$$

C. Domain with two semi-infinite directions and one fully unbounded BC.

$$\phi_{ref}(x,y,z) =$$

$$\left[\exp\left(10\left(1-\frac{1}{1-\left(\frac{2x-1.4L}{L}\right)^2}\right)\right)+\exp\left(10\left(1-\frac{1}{1-\left(\frac{2x-2.6L}{L}\right)^2}\right)\right)\right]$$

$$\exp\left(10\left(1-\frac{1}{1-\left(\frac{2y}{L}-1\right)^2}\right)\right)$$

$$\left[\exp\left(10\left(1-\frac{1}{1-\left(\frac{2z-0.6L}{L}\right)^2}\right)\right)-\exp\left(10\left(1-\frac{1}{1-\left(\frac{2z+0.6L}{L}\right)^2}\right)\right)\right].$$

APPENDIX C

CONVERGENCE FOR THE BIO-SAVART SOLVER

This section provides the convergence results for the finite difference approximation of order 2 and 6 in the case of the Bio-Savart solver presented in Section V.

Figure A: Convergence of the Biot-Savart solver using 2nd order differentiation `CHAT2` (—•—), `HEJ2`(—•—), `HEJ4`(—•—), `HEJ6`(—•—), `HEJ8`(—•—), `HEJ10`(—•—), `HEJ0`(—•—)

APPENDIX D

PERFORMANCE METRICS FOR WEAK AND STRONG SCALABILITY

A software running for T seconds on P resources in parallel can be characterized by the percentage of the time spent in parallel regions, α, and the percentage spent in serial regions, $\beta = 1 - \alpha$ such that $T = \alpha T + (1-\alpha)T$. When going from P_0 resources to P_1 with $r = P_1/P_0$ the ratio between resources, only the parallel regions will benefit from the gain and the execution time becomes

$$T_1 = \frac{\alpha T_0}{r} + (1-\alpha)T_0 . \tag{e}$$

Amdahl's law [22] defines the *speedup* as the ratio of both measured times:

$$s_P = \frac{T_0}{T_1} = \frac{1}{\frac{\alpha}{r}+(1-\alpha)} , \tag{f}$$

and uses s_p as a performance metric. This approach is usually referred to as strong scaling.

Figure B: Convergence of the Biot-Savart solver using 4th order differentiation `CHAT2` (—•—), `HEJ2`(—•—), `HEJ4`(—•—), `HEJ6`(—•—), `HEJ8`(—•—), `HEJ10`(—•—), `HEJ0`(—•—)

However, most practical applications scale the problem size with the available resources. In such a context, the time spent on P_1 resources scales with r (as the problem size scales with r) while only the parallel region benefits from the additional resources:

$$T_1 = r\left[\frac{\alpha T_0}{r}+(1-\alpha)T_0\right] = \alpha T_0 + r(1-\alpha)T_0 . \tag{g}$$

Therefore serial regions will lead to a longer execution time, while time spent in parallel regions will remain constant. This line of thought, usually called the weak scalability, leads to Gustafson's law [23] which defines the *efficiency* as the ratio of both times

$$\eta_{P,w} = \frac{T_0}{T_1} = \frac{1}{\alpha+r(1-\alpha)} . \tag{h}$$

We note that an equivalent efficiency can also be obtained from Amdahl's law as the ratio between the speedup s_P and theoretical gain that should have been obtained during the strong scalability, r:

$$\eta_{P,s} = \frac{s_P}{r} = \frac{1}{\alpha+r(1-\alpha)} . \tag{i}$$

Finally similar expressions can be obtained for $\beta = 1 - \alpha$, the serial percentage of the software. We also highlight that both the strong and the weak scalability are driven by quality of implementation which can be measured by β. A perfect scalability would lead to $\beta = 0$ and a perfect parallelization of the software.

APPENDIX E

DETAILS ON THE TESTCASES

A. Comparison with `accFFT`

Table A details the layout used for the comparison with `accFFT` described in Section VI-A.

B. Weak scalability

Table B details the layout used for the weak scaling analysis described in Section VI-C. The numbers in the table

nodes	P_x	P_y	P_z
1	1	8	16
2	1	16	16
8	1	32	32
32	1	64	64
128	1	128	128

Table A: Process distribution for the comparison with `accFFT` in Section VI-A

represents the process distribution in the three dimensions, the number of unknowns per process is kept constant to 96^3.

nodes	P_x	P_y	P_z
1	4	4	8
2	4	8	8
4	8	8	8
8	8	8	16
16	8	16	16
32	16	16	16
64	16	16	32
128	16	32	32
256	32	32	32
384	32	32	48

Table B: Process distribution for the weak scaling analysis of Section VI-C

Figure C: `isr`: Comparison of three different architectures, Vega (▮▪), MeluXina (▮▪) and Lumi (▮▪). Timings of weak scalability tests, performed with 96^3 unknowns per rank on a fully unbounded domain.

C. Strong scalability

Table C details the layout used for the strong scaling analysis described in Section VI-D. The numbers in the table represents the process distribution in the three dimensions, the total number of unknowns is kept constant to 1280^3.

nodes	P_x	P_y	P_z
1	4	4	8
2	4	8	8
4	8	8	8
8	8	8	16
16	8	16	16
32	16	16	16
64	16	16	32
128	16	32	32
256	32	32	32
384	32	32	48

Table C: Process distribution for the strong scaling analysis of Section VI-D

APPENDIX F

COMPARISON OF MAIN EUROPEAN SYSTEMS

This section includes additional time-to-solution and weak efficiencies for the `isr` and the `a2a` version of flups on different European systems, summarized in Table D. The results as presented in Figure C and Figure D for the `isr` version and Figure E and Figure F for the `a2a` implementation. For reference, we also provide the throughput per rank in Table E, Table F and Table G. The numbers have been multiplied by the factor 3/14 for better comparison with the other results.

Figure D: Implementation `isr`: Weak efficiency $\eta_{P,w}$ on Vega (●), MeluXina (●), and Lumi (●). Weak scalability tests performed with 96^3 unknowns per rank on a fully unbounded test case.

Name	Location	CPU	Interconnect	Transport Layer	OSU latency
Lumi	Finland	AMD EPYC 7763	200 Gb/s Slingshot-11	`libfabric 15.0.0 - CXI`	2.05 μs
MeluXina	Luxembourg	AMD EPYC 7H12	200 Gb/s Infiniband HDR	`ucx 1.13.1`	1.45 μs
Vega	Slovenia	AMD EPYC 7H12	100 Gb/s Infiniband HDR	`ucx 1.13.1`	1.99 μs

Table D: List of systems used for scalability testing

N nodes	1	2	8	64	128
Vega	33.82	31.62	25.50	23.09	17.11
MeluXina	32.14	31.54	29.59	29.19	26.57
Lumi	34.43	33.13	28.40	24.15	21.12

Table E: Implementation `nb`: throughput per rank [MB/sec] for a solve on the three parallel architectures. To account for the domain-doubling technique for unbounded BCs a normalization factor of 14/3 has been applied.

N nodes	1	2	8	64	128
Vega	42.16	38.88	28.55	21.13	16.29
MeluXina	40.94	38.39	32.49	26.92	23.82
Lumi	38.57	34.79	28.90	24.09	20.73

Table F: Implementation `isr`: throughput per rank [MB/sec] for a solve on the three parallel architectures. To account for the domain-doubling technique for unbounded BCs a normalization factor of 14/3 has been applied.

Figure E: `a2a`: Comparison of three different architectures, Vega ( ), MeluXina ( ) and Lumi ( ). Timings of weak scalability tests, performed with 96^3 unknowns per rank on a fully unbounded domain.

Figure F: Implementation `a2a`: Weak efficiency $\eta_{P,w}$ on Vega ( ), MeluXina ( ), and Lumi ( ). Weak scalability tests performed with 96^3 unknowns per rank on a fully unbounded test case.

N nodes	1	2	8	64	128
Vega	31.58	29.28	22.46	18.11	14.35
MeluXina	30.34	29.85	27.21	25.94	24.33
Lumi	34.19	33.96	28.41	22.93	16.05

Table G: Implementation `a2a`: throughput per rank [MB/sec] for a solve on the three parallel architectures. To account for the domain-doubling technique for unbounded BCs a normalization factor of 14/3 has been applied.