ETH

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Declaration of originality

I hereby confirm that I am the sole author of the written work here enclosed and that I have compiled it in my own words. Parts excepted are corrections of form and content by the supervisor.

Title of work (in block letters):

A memory-saving, type-independent
improvement to an open boundary FFT
solver in IPPL 2.0

Authored by (in block letters):
For papers written by groups the names of all authors are required.

Name(s):	**First name(s):**
Montanaro	Veronica

With my signature I confirm that
- I have committed none of the forms of plagiarism described in the 'Citation etiquette' information sheet.
- I have documented all methods, data and processes truthfully.
- I have not manipulated any data.
- I have mentioned all persons who were significant facilitators of the work.

I am aware that the work may be screened electronically for plagiarism.

Place, date	**Signature(s)**
Zurich, 12.09.2023	Veronica Montanaro

For papers written by groups the names of all authors are required. Their signatures collectively guarantee the entire content of the written paper.

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

PAUL SCHERRER INSTITUT

A memory-saving, type-independent improvement to an open boundary FFT solver in IPPL 2.0

Master Thesis

in Computational Sciences and Engineering

Department of Mathematics

Eidgenössische Technische Hochschule (ETH) Zürich

written by

Veronica Montanaro

supervised by

Dr. A. Adelmann (ETH Zürich)

scientific advisors

Dr. M. Frey (University of St. Andrews)
S. Mayani (Paul Scherrer Institute)
Dr. S. Muralikrishnan (Forschungszentrum Jülich)
Dr. M. Stoyanov (Oak Ridge National Laboratory)
Prof. S. Tomov (University of Tennessee)

September 12, 2023

Abstract

Particle In Cell (PIC) schemes are one of the most well-known methods in computational physics for a wide variety of applications, like astrophysics and particles kinetics in plasmas. Because of their large usage, many efforts have been made to make them suitable for heterogeneous architecture and High Performance Computing (HPC) resources. In this scenario, having an efficient solver for every set of the field equations is of vital importance. In this thesis we consider the electrostatic, collision-less field equations with open boundary conditions. For this case, Vico et al. (2016) have developed a Fast Fourier Transform (FFT) based algorithm that would achieve spectral accuracy [1], overcoming the limitations of the current state-of-the-art method presented by Hockney and Eastwood in [2]. However, the Vico method presents a major in-built flaw of requiring a high memory occupation. This is problematic especially within HPC frameworks, especially on multi-GPU simulations. Since GPUs have usually less memory than CPUs, to fully exploit these powerful units it's important to consider ways of fitting larger size problems without using too much memory. The goal of this thesis is to fix the memory flaw of the implementation of this algorithm within the Independent Parallel Particle Layer (IPPL) framework [3], and additionally to make the whole framework type independent. IPPL contains the particle and field operations of the Open Particle Accelerator Library (OPAL) [4], a parallel, open source and hardware portable tool, designed to run on any kind of architecture. We will use a theoretical trick that exploits the properties of a class of Real to Real (R2R) symmetric Fourier transforms, i.e. Discrete Cosine Transforms (DCTs). For this reason, the Discrete Cosine Transform (DCT) type needed for the solver will be implemented for GPUs inside the Highly Efficient FFT library (heFFTe) [5] library, used in IPPL for the FFT operations. Combining all these steps, at the end of the thesis we will end up having a memory-optimized, type-independent, scalable and performance portable solver, capable of running exascale machines and with an accuracy improvement with respect to the state-of-the-art free space Poisson solvers in the PIC context.

Contents

List of Figures

List of Tables

List of Listings

Acronyms

ALPINE	A pLasma Physics mINiapp for Exascale
C2R	Complex to Real
CPUs	Central Processing Unit
DCT	Discrete Cosine Transform
DCT-I	Discrete Cosine Transform of Type I
DCTs	Discrete Cosine Transforms
DFT	Discrete Fourier Transform
ECP	Exascale Project
FFT	Fast Fourier Transform
FLOPs	Floating-point Operations
GPUs	Graphic Processing Units
GPUs	Graphic Processing Unit
heFFTe	Highly Efficient FFT library
HPC	High Performance Computing
ICL	Innovative Computing Lab
IPPL	Independent Parallel Particle Layer
OPAL	Object Oriented Particle Accelerator Library
OPAL	Object Oriented Particle Accelerator Library
PIC	Particle In Cell
PSI	Paul Scherrer Institute
R2C	Real to Complex
R2R	Real to Real
UTK	University of Tennessee

Chapter 1

Introduction

The post-pandemic years have been dominated by extreme climate events, limited access to resources and increasing prices of energy. Policies to counter the climate crisis, as well as energy-saving regulations, are in constant development. In such a historical frame, what has been predicted about the future of microprocessors has been further confirmed: it is not sustainable to increase computing power of single-processors architectures. On the other hand, we need powerful machines for research in many different fields. Modelling of cosmology, plasma physics, weather or disease spreading are just few examples of the simulations that need fast, efficient and powerful architectures, capable of supporting simulations with higher resolutions. In this setting, Graphic Processing Units (GPUs) have become extremely attractive for scientific applications [9]. For HPC, these devices have made heterogeneous architectures, based on having different Central Processing Unit (CPUs) and GPUs in the same machine, possible. This solution provides both energy efficiency and power, allowing a higher number of Floating-point Operations (FLOPs) than single-processor machines.

Heterogeneous computing architectures are unavoidably moving towards the era of exascale computing, *i.e.* they're close to achieving a total of $\sim 10^{18}$ FLOPs. Currently, the US department of Energy has made the strategic move of opening and financing the Exascale Project (ECP). Computing nodes are being built with ever increasing depth of hierarchy. Hardware as well as performance portability, defined as *"a measurement of an application's performance efficiency for a given problem that can be executed correctly on all platforms in a given set"* [10], are key capabilities to make efficient use of them.

PIC methods are the method of choice in computational simulations of many physical applications including but not limited to particle accelerators, nuclear fusion and astrophysics. Hence, portability in the context of PIC schemes is the need of the hour to carry out these extreme scale simulations in current and next generation architectures. With IPPL [6] we have developed a performance portable framework for PIC methods.

Especially for PIC codes, another feature that has revealed itself particularly useful for the future is the availability of mixed precision. For code with large number of operations, especially on GPUs, one must carefully consider that double precision data occupies more memory, and double precision operations in general are more expensive. As this could possibly trump the benefits of running code on hybrid architectures, single precision operations and data structures are preferrable. However, a lower number of floating-point digits may lead to larger errors. For this reason, it has become a priority of various developers of tools for scientific simulations to allow type-independency, in order to provide the user with the possibility of selecting the best configuration of single and double precision data structures and operations. Furthermore, one also needs algorithms which provide good accuracy with single precision types.

In the context of electrostatic PIC, Vico et. al [1] suggest a fast algorithm for computing volume potentials, which can be a useful method for preconditioners and solvers, which converges spectrally, i.e. faster than any fixed order of the number of grid points, for smooth enough functions. This attractive feature fulfils the request of ensuring the best possible accuracy for any precision. The algorithm has already been implemented by [3] in IPPL in a performance portable and parallel manner. Since coarser grids are required to achieve similar accuracy as current state-of-the-art solvers for the Poisson equation, the memory footprint could be significantly reduced to fit the problem on Graphic Processing Unit (GPUs). This is then also a good test case for single precision runs of IPPL.

Chapter 2

Theory

In this chapter, the theoretical background needed to understand the algorithms and results from the thesis will be stated.

Computational discretization notation

Throughout this section, we will work with computational grids, thus a brief introduction on the notation used is needed. Since we will work with three dimensions, we will use L_d, with $d = x, y, z$, to indicate the length of a physical domain defined on $[0, L_d]$ in the d-th dimension. The number of points of the computational grid to be superimposed on the same dimension will be referred as N_d. We will use uniform grid geometries, such that all the points in the d direction are equally spaced with distance $h_d = \frac{L_d}{N_d}$. Given a physical quantity on the domain, for example a 3-dimensional vector $p(x, y, z)$, we will use the indices $p_{i,j,k}$ with $i, j, k \in [0, 1, \ldots N_x - 1], [0, 1, \ldots N_y - 1], [0, 1, \ldots N_z - 1]$ to indicate that p is evaluated at coordinates $x = ih_x, y = jh_y, z = kh_z$.

2.1 Particle In Cell

PIC methods refer to a family of methods for plasma modelling and simulation [11, 12] . These methods use the kinetic approximation of a plasma, prioritizing the interaction between particles and fields. In this approximation, the distribution of particles is modeled as a set of macropartiicles. The case that will be observed in this section is the one of a collision-less plasma in an electrostatic field. In the domain where the macroparticles lie, a spatial grid is superimposed (Figure 2.1) so that the particle's charges are interpolated on its grid points, in order to obtain the charge density ρ. This phase is usually called *scatter.* After the scatter phase, for every grid point represented by the positional vector $\vec{x} \in \mathbb{R}$, the electric potential $\phi(\vec{x})$ and the electric field $\vec{E}(\vec{x})$ need to be obtained. The field equations at $\vec{x}$ are given by:

$$\nabla^2 \phi(\vec{x}) = -\frac{\rho(x)}{\epsilon_0}, \tag{2.1}$$

$$\vec{E}(\vec{x}) = -\vec{\nabla} \phi(\vec{x}), \tag{2.2}$$

where ρ is the charge density field and ϵ_0 is the dielectric constant in vacuum. (2.1) is a Poisson equation, and throughout the work, when such equation will be mentioned, the electrostatics one will be assumed.

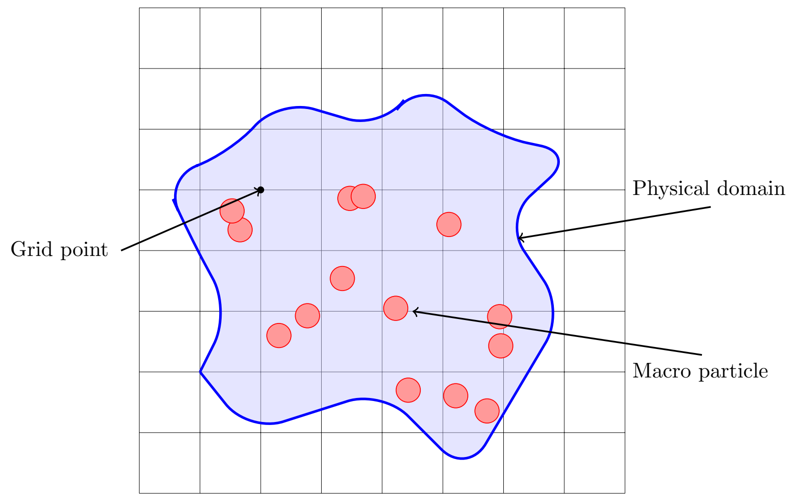


Figure 2.1: Super-imposition of a computational grid over a 2-D physical domain populated by macroparticles.

After the field equations are solved on the grid, the field values are interpolated back on the particles (*gather*) to compute the new forces acting on them at the current timestep. Only then, the equations of motion are updated with a timestepping scheme (ususally a leapfrog scheme gives enough accuracy), and particles are *pushed* on the grid accordingly.

For a non-relativistic regime, the internal magnetic field B is zero, so the equations of motion for a particle with position $|\vec{r}|$ are given by:

$$\frac{d\vec{r}}{dt} = \frac{\vec{p}}{m\gamma} \tag{2.3}$$

$$\frac{d\vec{p}}{dt} = -q(\vec{E} + \vec{E}_{ext} + \vec{v} \times \vec{B}_{ext}), \tag{2.4}$$

where q, m are respectively the charge and the mass of the particle, and γ is the Lorentz factor. Here, $\vec{E}$ is the internal electric field coming from grid contributions, while $\vec{E}_{ext}$ and $\vec{B}_{ext}$ are external source terms.

Figure 2.2 summarizes the steps of the electrostatic PIC scheme without collisions.

2.2 FFT Poisson Solvers for open boundary Poisson's equation

Finding a way to efficiently solve the field equations in a computer simulation is crucial, and a lot of methods have been developed for the sake of it. This section analyzes the case for the Poisson equation, used in electrostatics PIC codes. A widely studied set of solvers are those based on Fourier spectral methods, used to accurately solve Poisson equations on rectangular domains for different kinds of boundary conditions. Computationally, they present a lot of advantages with respect to other methods such as finite differences. First of all, they require fewer unknowns, and thus less computation time and memory occupation. Moreover, those methods achieve

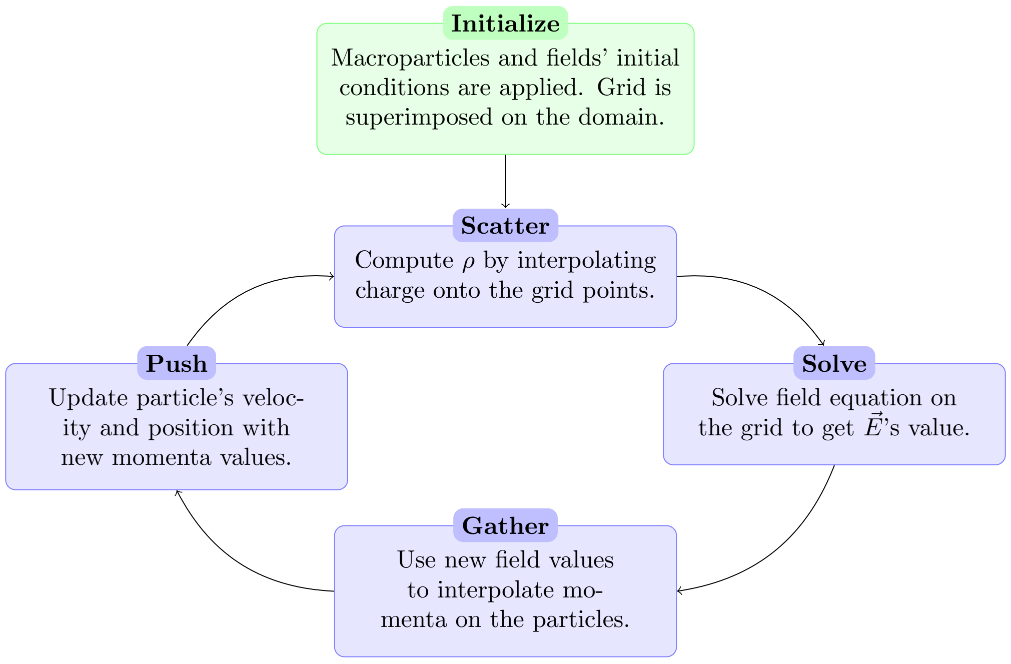

Initialize
Macroparticles and fields' initial conditions are applied. Grid is superimposed on the domain.
Scatter
Compute ρ by interpolating charge onto the grid points.
Solve
Solve field equation on the grid to get $\vec{E}$'s value.
Gather
Use new field values to interpolate momenta on the particles.
Push
Update particle's velocity and position with new momenta values.

Figure 2.2: Collision-less electrostatic PIC loop scheme.

faster convergence than finite differences, since their convergence rate is only determined by the regularity of the underlying function [13, Chapter 1]. This behavior is referred in literature as spectral accuracy. Fourier methods are particularly suited for problems with periodic boundary conditions [13, Chapter 2], but for other types of boundaries periodicity can be induced. This will be the main intuition behind Hockney and Eastwood's [2] solver, taken into analysis in the next sections. The authors make use of the fact that for (2.1) with open boundary conditions (meaning that $\phi(\vec{x}) \to 0$ as $|\vec{x}| \to \infty$), the potential ϕ can be written as:

$$\phi(\vec{x}) = \int G(\vec{x} - \vec{x'})\rho(\vec{x'})d\vec{x'} = \{G * \rho\}(x), \tag{2.5}$$

where $*$ is the convolution operator, and $G(\vec{x}) = -\frac{1}{4\pi|\vec{x}|}$ for $\vec{x} \in \mathbb{R}^3$, is called *Green's function of the interaction* [14] and represents the impulse response for (2.1), *i.e.* the solution of the equation for $\rho(\vec{x}) = \delta(\vec{x})$. Thanks to this property, we now can make use of the following theorem from signal theory:

Theorem 1 (Convolution Theorem)**.** *Given two functions $f(x), h(x)$ defined on $[0, \infty)$, and defining $\mathcal{F}\{\}$ as the Fourier transform operator, which computes:*

$$\mathcal{F}\{f\}(s) = \int f(x)e^{-2i\pi xs}dx. \tag{2.6}$$

Then, calling $q(x) = \{f * h\}(x)$ the convolution operation between the two functions, we have:

$$\mathcal{F}\{f * h\}(s) = \mathcal{F}\{f\}(s)\mathcal{F}\{h\}(s). \tag{2.7}$$

As a consequence of Theorem 1, the folllowing corollary is constructed:

Corollary 1 (Corollary to the Convolution Theorem). *Defining $\mathcal{F}^{-1}\{\}$ as the inverse Fourier transform operator, which computes*

$$\mathcal{F}^{-1}\{F\}(x) = \int f(x) e^{-2i\pi xs} dx, \tag{2.8}$$

the convolution between two functions $f(x)$ and $h(x)$, defined in the same way as Theorem 1, will be equivalent to:

$$\{f * h\} = \mathcal{F}^{-1}\{\mathcal{F}\{f\}\mathcal{F}\{h\}\}. \tag{2.9}$$

It can be easily seen that by applying Corollary 1 to (2.5), finding the potential ϕ for solving (2.1) can be reduced to solving the following equation:

$$\phi(\vec{x}) = \mathcal{F}^{-1}\{\mathcal{F}\{G\}\mathcal{F}\{\rho\}\}. \tag{2.10}$$

The reason why this is relevant is because Fourier transforms can be easily discretized on a machine, and several well-studied methods to compute them fast and efficently exist [15, 16]. Thus, a method that involves them is preferrable to integral computation methods that would be needed to evaluate (2.5).

A Fourier transform for a discrete-time signal is called a Discrete Fourier Transform (DFT). The next paragraphs will be dedicated to a short introduction on it and on convolution in discrete space, since it will be relevant later on in the work.

Given an input signal $x = \{x_0, x_1, ..., x_{N-1}\}$, with $x_{n\in\{0,N-1\}} \in \mathbb{C}$, we define its DFT as the transform that computes the output signal $Y = \{Y_0, Y_1, ..., Y_{N-1}\}$, with $Y_{k\in\{0,N-1\}} \in \mathbb{C}$, such that its coefficients are:

$$Y_k = \frac{1}{N}\sum_{n=0}^{N-1} x_n e^{\frac{-2\pi i k n}{N}}, \tag{2.11}$$

where i is the imaginary unit, and the inverse transform is defined as:

$$x_k = \frac{1}{N}\sum_{n=0}^{N-1} Y_n e^{\frac{2\pi i k n}{N}}. \tag{2.12}$$

(2.11) is equivalent to (2.6) for a discrete signal, and the same can be said for (2.12) and (2.8). A DFT is a unitary transform, meaning $DFT^{-1}\{DFT\{x\}\} = x$. For a d-dimensional array x_n, where $n = (n_1, n_2, \dots, n_d)$, (2.11) and (2.12) can be rewritten with $\frac{1}{N} = \frac{1}{N_1}\frac{1}{N_2}\cdots\frac{1}{N_d}$ and the sum operator as $\sum_{n_0=0}^{N_1-1}\sum_{n_1=0}^{N_2-1}\cdots\sum_{n_d=0}^{N_d-1}$.

A DFT computed naively has a complexity of $O(N^2)$, with $N = N_1 N_2 N_3 \dots N_d$, however nowadays most of the algorithms that compute it are derived from the FFT algorithm [17, 16], which instead has a computational complexity of $O(N \log N)$.

The conclusion to be drawn is that (2.10)'s complexity can be reduced to $O(N \log N)$ with $N = N_x N_y N_z$ in the 3-dimensional case. This is a significant reduction compared to brute force convolution in this discrete space, evaluated as

$$\phi_{i,j,k} = h_x h_y h_z \sum_{i'=0}^{N_x-1}\sum_{j'=0}^{N_y-1}\sum_{k'=0}^{N_z-1} G_{i-i',j-j',k-k'}\rho_{i',j',k'}, \tag{2.13}$$

since such computation would require a total complexity of $O(N^2)$.

In the next sections two algorithms that use this intuition and that are currently implemented in IPPL will be introduced. First is the Hockney-Eastwood method [2], which is the current state-of-the-art for open boundary Poisson solvers. After it, an improvement to said method developed by Vico et al. in 2016 [1] will be presented.

2.3 Hockney-Eastwood method

From now on, we will consider a physical domain in three dimension of length $[0, L_x] \times [0, L_y] \times [0, L_z]$ with open boundary conditions in all directions, and a computational grid of size $N_x \times N_y \times N_z$. Hockney's algorithm replaces (2.13) with a cyclic convolution, which requires the construction of the functions ρ_2 and G_2 derived from ρ and G, such that they're both periodic with the same period.
The steps are the following as in [18]

1. Create an extended mesh, doubling all the directions where the boundary conditions are non-periodic. In our case, the computational size will be $2N_x \times 2N_y \times 2N_z$
2. Restrict the charge distribution to the bottom-left corner $[0, N_x-1]\times[0, N_y-1]\times[0, N_z-1]$, resulting in a zero-padded region for $[N_x, 2N_x-1] \times [N_y, 2N_y-1] \times [N_z, 2N_z-1]$. This step corresponds to the construction of the periodic charge distribution ρ_2 such that:

$$\rho_2(\vec{x}) = \begin{cases} \rho(\vec{x}) & if\ \vec{x} \in [0, L_x) \times [0, L_y) \times [0, L_z) \\ 0 & otherwise. \end{cases} \tag{2.14}$$

3. Construct the periodic Green's function such that:

$$G_2(\vec{x}) = \begin{cases} G(\vec{x}) & if\ \vec{x} \in [0, L_x) \times [0, L_y) \times [0, L_z) \\ G(2L_x - x, y, z) & if\ \vec{x} \in [L_x, 2L_x) \times [0, L_y) \times [0, L_z) \\ G(x, 2L_y - y, z) & if\ \vec{x} \in [0, L_x) \times [L_y, 2L_y) \times [0, L_z) \\ G(x, y, 2L_z - z) & if\ \vec{x} \in [0, L_x) \times [0, L_y) \times [L_z, 2L_z) \\ G(2L_x - x, 2L_y - y, z) & if\ \vec{x} \in [L_x, 2L_x) \times [L_y, 2L_y) \times [0, L_z) \\ G(2L_x - x, y, 2L_z - z) & if\ \vec{x} \in [L_x, 2L_x) \times [0, L_y) \times [L_z, 2L_z) \\ G(x, 2L_y - y, 2L_z - z) & if\ \vec{x} \in [0, L_x) \times [L_y, 2L_y) \times [L_z, 2L_z) \\ G(2L_x - x, 2L_y - y, 2L_z - z) & if\ \vec{x} \in [L_x, 2L_x) \times [L_y, 2L_y) \times [L_z, 2L_z) \\ -\frac{1}{4\pi} & if\ \vec{x} = 0 \end{cases} \tag{2.15}$$

   The last term is a regularization term chosen to replace the singularity of the Green's function at $\vec{x} = 0$.
4. Compute the extended potential $\phi_2(\vec{x})$ by plugging ρ_2 and G_2 in (2.10).
5. ϕ_2 is non-zero and equal to the exact potential ϕ for $\vec{x} \in [0, L_x) \times [0, L_y) \times [0, L_z)$ [18]. So, by restricting it we end up with the physical solution to the Poisson's equation.

The storage required for the computation of G in the Hockney method is of $2^d N$ for a d-dimensional boxed domain, where $N = N_{x_1} N_{x_2} \dots N_{x_d}$. The number of operations needed is around $\sim O((2^d N)\log\left(2^d N\right))$. However, even if it is proved that the circular convolution used in step 4 of the algorithm approximates correctly (2.5), the method presents still some limitations. One has to take into account that for discretizing the integral of the Fourier transformation (2.5), the numerical method used is the midpoint rule, which is limited to second order of convergence (so, for mesh spacing h, we should expect the algorithm to converge like $O(h^2)$). Moreover, the choice of regularizing $G(0)$ with a finite value also influences the convergence, as explained in [19, 20].

2.4 Vico-Greengard method

The method presented by Vico, Greengard et al. makes use of a trick to get rid of the second-order convergence bound that the Hockney method suffers from. The key difference in their algorithm lies in the regularization of the Green's function. It can be proved (full proof can be found in [1]) that for the 3-dimensional Green's function defined as in the previous section, we obtain the following function in Fourier's space, which is a spectral representation of an equivalent Green's function to the real-space one $G = \frac{1}{4\pi|\vec{x}|}$:

$$G_0^L(\vec{s}) = \mathcal{F}\{G\}(\vec{s}) = 2\Big(\frac{\sin\left(L|\vec{s}|/2\right)}{|\vec{s}|}\Big)^2, \tag{2.16}$$

where $\vec{s}$ is the Fourier wave vector, and $L > \sqrt{d}$ (usually taken as $L = 1.1\sqrt{d}$) is the length of the truncation window. This spectral representation of an equivalent G is regular for $G(0)$, unlike G_2 that needed a regularization term. Because of this result, $\mathcal{F}\left\{G\right\}(\vec{s})$ can be pre-computed directly in Fourier's space, and the authors show that, since G_0^L is an analytic function, the computation of the integral via trapezoidal rule can compute ϕ with higher accuracy, leading to spectral convergence. This function requires a mesh of size $4N_x 4N_y 4N_z$, in order to be sufficiently sampled. The first factor of two comes from the same circular convolution factor as Hockney's method, and the latter originates from the oscillatory behaviour of G_0^L. This is a consequence of the Nyquist's sampling theorem, which states that a function can be accurately reconstructed without aliasing by sampling it at frequency $f_s \geq 2B$, where B is the highest frequency in the function's spectrum.

The first two steps of the method, as well as the last two, are the same as the Hockney method. What changes are the central steps: we already have G_0^L pre-computed on the $4N_x 4N_y 4N_z$ mesh, so we can compute $\mathcal{F}^{-1}\left\{G_0^L\right\}$ and restrict it to the $2N_x \times 2N_y \times 2N_z$ mesh, obtaining G_2. After this step, we can proceed with the computation of ϕ_2 and its restriction on the physical domain to obtain ϕ.

Although the method gives better accuracy for the same grid spacing and mesh points of the Hockney method (for a smooth enough ρ), previous scaling studies [3] show that the additional memory requirement to store the $4N_x 4N_y 4N_z$ mesh can bring to significant disadvantages. In fact, this algorithm does not scale with some configurations on GPUs since large problem sizes do not fit, and thus the time spent on communication dominates over computation time.

As an example, in the IPPL framework, this extra data structure leads to an overall multiplication factor in memory of $\sim [2.5 - 3]$ (depending on the choice of single or double precision), with respect to Hockney. Details for this calculation can be seen in the appendix.

In order to fully exploit exascale with this algorithm, investigations on possible memory reduction tricks must be carried.

2.4.1 Algorithmic improvements to Vico-Greengard method

One possible solution is described in the conclusions of [3], and its implementations and results will be the core of this work. The idea is to exploit the symmetric properties of G_0^L. It can be seen easily that the function is even-symmetric around $|\vec{s}| = 0$ and, since G_2 is periodic, the following theorem holds:

Theorem 2. *Given a function $f(x) \in R$ and its Fourier transform $F(s) = \mathcal{F}\{f\}$, then $F(s)$ satisfies Hermitian symmetry. Namely:*

$$F(s) = F^*(-s). \tag{2.17}$$

Furthermore, if $f(x)$ is periodic, the Hermitian symmetry is satisfied if and only if $Im(F(s)) = 0 \; \forall s \implies F \in R$.

The theorem also holds for discrete-time transforms. So, for an input signal $x = \{x_0, x_1 \ldots, x_{N-1}\}$ with $x_{n\in\{0,\ldots,N-1\}} \in \mathbb{R}$, then it can be proven that its DFT has Hermitian symmetry, namely $Y_k = Y^*_{N-k}$. If we also take x to be periodic, *i.e.* $x_0 = x_{N-1}$, then we can see that the Hermitian symmetry is satisfied if and only if $Y \in \mathbb{R}$.

Because of the unitary nature of the DFT, if the input vector x is real, even and periodic, then its transform Y will also be real. For this case, it can be shown that the DFT is equivalent to another family of transforms, called DCTs [21, 7, 22]. These transforms take as an input the signal composed of only the non-repeating terms of the sequence, such as the result is equivalent to a DFT of twice the length. DCTs are defined differently depending on where the symmetry is centered around, see table 2.1. In this work, the transform taken into analysis will be the Discrete Cosine Transform of Type I (DCT-I). Its definition is the following:
let x be a collection of real coefficients of size M, such that $x_0 = x_M$ and, in general $x_{n\in\left\{0,1,\ldots\lfloor\frac{M}{2}\rfloor\right\}} = x_{M-n}$, and let $N = \lfloor\frac{M}{2}\rfloor$ be the number of non-repeating coefficients appearing in the first half of the signal. Then transforming x with a DCT-I will give a real output signal Y of size N, with the k-th coefficient computed as:

$$Y_k = \frac{1}{2(N-1)}\left(x_0 + (-1)^k x_{N-1} + 2\sum_{n=1}^{N-2} x_n \cos\left[\frac{\pi k n}{N-1}\right]\right). \tag{2.18}$$

Because of the unitary nature of the DFT, it can be shown that $\{\text{DCT-I}\}^{-1} = \text{DCT-I}$.

Having introduced this family of real-to-real symmetric transforms, the connection with the Vico algorithm becomes immediate. We can avoid constructing the larger grid by pre-computing only the first half of the real coefficients of G_0^L on a grid of size $2(N_x+1)\times 2(N_y+1)\times 2(N_z+1)$ (the extra gridpoint is necessary since we also need $G_0^L(0)$), obtaining the function $G_{0,2}^L$. Performing $\text{DCT-I}^{-1}\left\{G_{0,2}^L\right\}$ will be equivalent of performing $FFT^{-1}\left\{G_0^L\right\}$, and will give G_2 as a result. All that is left after is to restrict G_2 from the grid of size $2(N_x+1)\times 2(N_y+1)\times 2(N_z+1)$ to $2N_x \times 2N_y \times 2N_z$, and proceed in the same way as we did before.

2.5 Relation between DCT and real-to-complex DFT

When it comes to implementing DCTs on a machine, it might be useful to study their relationship with the DFT defined in 2.11. Given a signal x of length N, a DCT-I of one periodic and

DCT type	Symmetry point 1		Symmetry point 2	
	Type of simmetry	location	Type of simmetry	location
I	Even	$n = 0$	Even	$n = N - 1$
II	Even	$n = -\frac{1}{2}$	Even	$n = N - \frac{1}{2}$
III	Even	$n = 0$	Odd	$n = N$
IV	Even	$n = -\frac{1}{2}$	Odd	$n = N - \frac{1}{2}$

Table 2.1: The table shows the symmetric properties of the different types of DCTs [7]. Every DCT shows two centers of symmetry, positioned either at the mid-points between two data points or at the data point itself. Note that N corresponds to the index of the last non-repeated data point.

even symmetric extension of it can be computed through a real-to-complex DFT. The steps are the following:

1. Construct the signal y, of length $4(N-1)$, such that:

$$y[n]_{n\in[0,\dots,4(N-1))} = \begin{cases} x[n/2], & \text{if } n \text{ even and } n \leq 2N \\ x[(4(N-1)-n)/2] & \text{if } n \text{ even and } 2N < n < 4(N-1) \\ 0.0 & \text{otherwise} \end{cases} \tag{2.19}$$

2. Perform a DCT on the extended signal, which will result in:

$$Y[k] = \frac{1}{4(N-1)} \sum_{n=0}^{4(N-1)-1} y_n e^{\frac{-2\pi i k n}{4(N-1)}} \tag{2.20}$$

$$= \frac{1}{4(N-1)} \left(e^0 x_0 + e^{-\pi i k} x_{N-1} + \sum_{n=1}^{N-2} x_n \left(e^{\frac{-2\pi i k 2n}{4(N-1)}} + e^{\frac{-2\pi i k (4(N-1)-2n)}{4(N-1)}} \right) \right) \tag{2.21}$$

$$= \frac{1}{4(N-1)} \left(x_0 + (-1)^k x_{N-1} + \sum_{n=1}^{N-2} x_n \left(e^{\frac{-\pi i k n}{N-1}} + e^{\frac{\pi i k n)}{N-1}} e^{-2\pi i k} \right) \right) \tag{2.22}$$

$$= \frac{1}{2(N-1)} \left(x_0 + (-1)^k x_{N-1} + 2 \sum_{n=1}^{N-2} x_n \cos\left[\frac{\pi k n}{N-1} \right] \right), \tag{2.23}$$

which proves that (2.20) for signal y = eq. (2.18) for signal x.

3. Because of the Hermitian symmetry property, extract the real part out of the non-repeating elements of Y, up to the k-th element, with $k = N - 1$.

Since the inverse of the DCT-I is itself, then the previous is also valid for an inverse DFT.

Chapter 3

Methodology

3.1 heFFTe

The heFFTe [5] provides a series of highly linearly scalable FFT algorithms, designed to target exascale machines. Its purpose makes it an extremely good candidate for exascale and performance portable applications, and in fact it constitutes the backbone of the FFT class of IPPL. In order to run on every architecture, it provides back-ends for both CPUs and GPUs. Said back-ends use vendor libraries for Fourier Transforms like FFTW [23], Nvdia's cuFFT [8] and Intel's MKL [24], which at the current state-of-the-art do not achieve scalability on many-core heterogeneous machines by themselves alone.

To match heFFTe's naming scheme, in the following we will refer to (2.11) as *forward* transform, and (2.12) as *backward* transform.
heFFTe is more than a wrapper around different back-ends. The rest of this section will go through the library's main features and strengths, in particular the decomposition and communication options provided. Multi-dimensional FFTs in heFFTe are performed through a series of reshape-and-compute operations along every dimension, where the reshapes are determined by the decomposition technique used. The library provides two: *pencils* and *slabs*. While the former performs three reshape operations and computes three 1-D transforms along the 3 spatial direction, the latter only goes through two reshape operations: the first one computes a 2-D transform, and the second one reshapes the third dimension in a pencil to perform a 1-D transform. A graphical representation can be seen in Figure 3.1.

In the reshaping, sending of data among MPI ranks is of vital importance. heFFTe gives the user the possibility to choose from different communication routines. The choice is between point-to-point and collective communication, with blocking (packing/unpacking routines and communication are separated by an MPI barrier) and non-blocking (packing and unpacking overlaps with communication) options (details on the type of routines in Table 3.1). For the reshape routine, the option of reordering the boxes constructing the domain between MPI ranks is also left to the user.

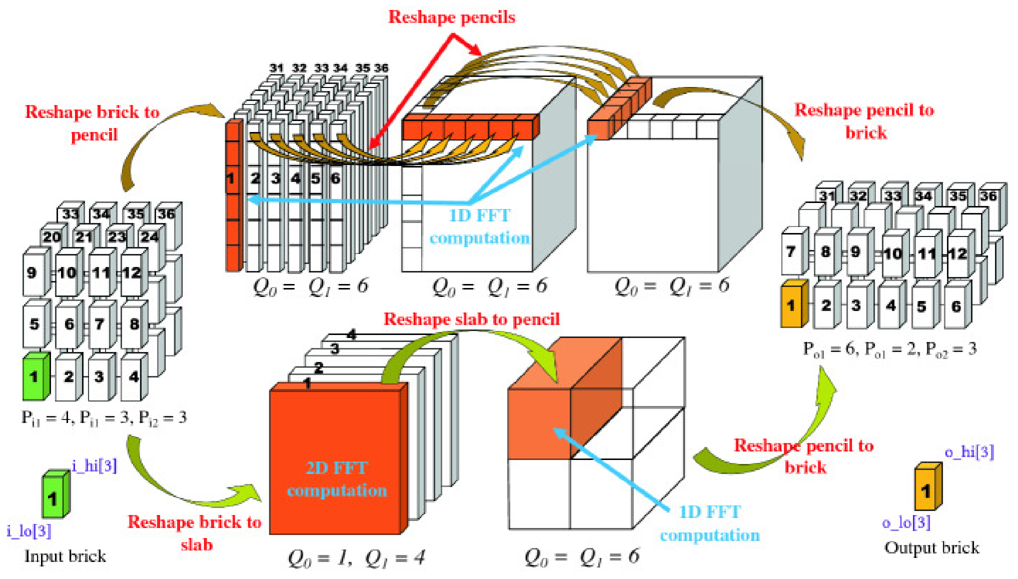


Figure 3.1: Schematics of heFFTe's decomposition options. The upper row shows the pencils algorithm, and the bottom the slabs algorithm. Initial and final processor grids are indicated as P_{i1}, P_{i2}, P_{i3} and P_{o1}, P_{o2}, P_{o3}. Original taken from Figure 2 of [5].

Table 3.1: MPI and native routines used by different communication options in heFFTe. Adapted from Table 1 of [5].

Point-to-point		Collective (All-to-all)	
Blocking	Non blocking (pipelined)	Blocking	Non blocking
`MPI_Send`	`MPI_ISend`	`MPI_Alltoallv`	`heFFTe_Alltoall`
`MPI_Recv`	`MPI_IRecv`	`MPI_Allreduce`	
`MPI_Sendrecv`		`MPI_Barrier`	

3.2 DCT-I Kernel implementation

Various signal processing libraries for high-level languages, like the ones provided by MATLAB or Python, include an in-built implementation of the different flavors of Real Even DFT i.e. DCTs [25, 26]. However, between the vendor libraries used by heFFTe, FFTW is the only one where an in-built implementation can be found [27, Section 4.8.2 and 4.8.3]. For the other back-ends, R2R transforms are implementented via means of the native Real to Complex (R2C) or Complex to Real (C2R) transforms, respectively for forward or backward transforms. To make sure that the result matches the one given by a DCT, there exist a `pre_post_processor` class which manipulates the input and output data. This is shown in detail in Algorithm 1 for the forward case. In our case, we implemented the DCT-I for the cuFFT backend to run on Nvidia GPUs following the steps (2.19) to (2.23) shown in section 2.5. The pre-processing kernel computes the signal from (2.19), then a cuFFT R2C (or C2R) is executed to match step (2.20), and finally the post-processing kernel extracts the real part as in (2.23). However, some adjustments had to be

FFT type	Input size	Output size
C2C	N *cufftComplex*	N *cufftComplex*
C2R	$\lfloor \frac{N}{2} \rfloor + 1$ *cufftComplex*	N *cufftReal*
R2C	N *cufftReal*	$\lfloor \frac{N}{2} \rfloor + 1$ *cufftComplex*

Table 3.2: Data layout for 1-D data used by cuFFT as of [8], showing expected sizes of input/output data. In the case of multidimensional transforms the size change is only applied to the n-th dimension.

made in order to adapt to cuFFT.

Algorithm 1 Computation of real-to-real transforms for non-FFTW back-ends. This example shows a forward transform, and swapping the complex conversion from output to input gives the backward transform process.

```
$inputOriginal \leftarrow$ data to be transformed
$inputExtended \leftarrow$ preProcessingKernel($input$)

$outputComplex \leftarrow$ transformR2C($inputOriginal$)

$outputReal \leftarrow$ postProcessingKernel($outputComplex$)
```

First, we had to take into account how cuFFT handles data sizes for C2R and R2C transforms. As table 3.2 shows, cuFFT takes into account the Hermitian symmetry to save memory when transforming, storing only non-redundant coefficients. This means that for a R2C transform, from an input of N real numbers, $\lfloor \frac{N}{2} \rfloor + 1$ complex numbers will be outputted, and vice versa in case of a C2R transform. Moreover, how complex number are stored in memory needs also to be considered. The way cuFFT handles a complex number is with a C vector of two floating point numbers, either in double, single of half precision. The real part is stored in the first number, and the complex part in the second. Therefore, an array of N complex numbers will result in a total of $2N$ total floating-point numbers to store, with the real parts located at the even indices, and the imaginary part located at odd indices.

The consequence of these technicalities are that the number of elements in the DCT and the number of numbers to be stored in memory, as well as the indices of the complex arrays will differ. This afffects the implementation of the pre- and post-processing kernels for the forward and backward transforms. Despite these being theoretically the same because of the unitarity of DCT-I (see Section 2.4.1), they show slight differences in practice. Listing 3.1 and listing 3.2 show the details of the implementations; note that both the forward and the backward post-processing kernels have the same implementations, but different logical meaning. The former extracts the real part of the resulting (complex) signal, while the latter retrieves the even elements of the real, extended signal obtained.

```
// DCT-I (REDFT00)
// even symmetry and periodicity; size 2N-1
//(a b c d) -> (a 0 b 0 c 0  d  0 c 0 b 0)
```

```
template<typename scalar_type>
__global__ void cos1_pre_forward_kernel(int N, scalar_type const *input,
    scalar_type *fft_signal){
    int ind = blockIdx.x*BLK_X + threadIdx.x;

    if(ind < N){
        fft_signal[2*ind] = input[ind];
        fft_signal[2*ind+1] = 0.0;
    }

    if(ind > 0 && ind < N){
        fft_signal[4*(N-1)-2*ind] = input[ind];
        fft_signal[4*(N-1)-2*ind+1] = 0.0;
    }
}

//extract real parts
//(c1 c2 c3) -> (c1.x c2.x c3.x)
template<typename scalar_type>
__global__ void cos1_post_forward_kernel(int N, scalar_type const *fft_signal,
    scalar_type *result){
    int ind = blockIdx.x*BLK_X + threadIdx.x;

    if(ind < N){
        result[ind] = fft_signal[2*ind];
    }
}
```

Listing 3.1: Pre and post processing CUDA kernels for the forward transform in cuFFT back-end.

```
// IDCT-I backward kernel for DCT-I. The transform itself doesn't change, since (
    DCT-I)^-1=DCT-I.
// However, the kernel has slight changes to adapt to the c2r transform.
// set imaginary parts to zero; even symmetry
// (a b c) -> (a,0 b,0 c,0 b,0 0,0)
template<typename scalar_type>
__global__ void cos1_pre_backward_kernel(int N, scalar_type const *input,
    scalar_type *fft_signal){
    int ind = blockIdx.x*BLK_X + threadIdx.x;

    if(ind < N){
        fft_signal[2*ind] = input[ind];
        fft_signal[2*ind+1] = 0.0;
    }

    fft_signal[4*(N-1)] = 0.0;
    fft_signal[4*(N-1)+1] = 0.0;

    if(ind > 0 && ind < N){
        fft_signal[4*(N-1)-2*ind] = input[ind];
        fft_signal[4*(N-1)-2*ind+1] = 0.0;
    }

}

// Extract even elements
// (a b c d e f) -> (a c e)
template<typename scalar_type>
__global__ void cos1_post_backward_kernel(int N, scalar_type const *fft_signal,
    scalar_type *result){
    int ind = blockIdx.x*BLK_X + threadIdx.x;

    if(ind < N){
        result[ind] = fft_signal[2*ind];
    }

}
```

Listing 3.2: Pre and post processing CUDA kernels for the backward transform in cuFFT back-end.

3.3 Improved Vico solver

After having implemented the DCT-I kernels for GPU back-ends, the steps described in Section 2.4.1 were incorporated in IPPL. For this purpose the class `FFTPoissonSolver.h`, which includes all the methods for the open-boundary Poisson Solver, was modified. It is now possible to call the optimized Vico solver by passing the string `VICO_2` to the class constructor.

Moreover, a change in the `FFT.h` class had to be done: during the work, we found out that the two FFTW and cuFFT transforms had a slight change in behavior due to how the two libraries handle normalization. In short, transforming backwards without normalization gave a different results for the two of them. This behavior was undiscovered before as the other cases (backwards with normalization, forward without normalization, which is the default for the heFFTe tests) carried out the same results, and a full transform cycle (forward and backward, with normalization in one of the two) would still lead to the initial data. For this reason, additional $\frac{1}{8}$ and 8 factors had to be added to the wrapper class for FFTW R2R forward and backward transform, respectively. Further correction of this behavior is currently in discussion to be incorporated in the library itself.

3.4 Mixed precision and type independency

Sometimes, for GPU runs single precision is preferred, for a variety of reasons. The main point is memory saving: since some GPUs don't go under machine precision of 10^{-6}, there is no need of having data occupying more memory if a lower precision cannot be achieved. Part of the thesis work was spent making IPPL type-independent. This implies a stronger meaning than just switching between double and single precision, since having freedom of choice for every data structure can allow the user to choose for which physical quantities each precision is required. A good criterion for these choices, for example, is to set everything that doesn't cause a drastic reduction in accuracy or unphysical behavour to single precision, and the rest in double. With the solver class `FFTPoissonSolver`, some extra carefulness was needed since these class contain a lot of type dependencies between member variables, so the main focus was finding a configuration that minimzes truncation errors and maximizes memory saving in case of mixed precision runs. The main modification in this class was the type of the Green's function G, since before it was hard-coded in double. The class is templated on two field types, `FieldRHS` for the charge density and `FieldLHS` for the electric field. The best choice turned out to be having the type of G be the same as `FieldRHS`. The reasoning behind this choice lays in the fact that, if G and ρ would be of different types, the convolution between them would lead to a truncation error that would propagate later when the result is assigned to $\vec{E}$. This means that, in case the type for the right hand side and left hand side fields differ we will encounter the minimal number of truncations (only happening during the assignment of $\vec{E}$). The other FFT solver for periodic boundary conditions, implemented in the class `FFTPeriodicPoissonSolver`, was more trivial to make type-independent since no other major dependencies are hidden inside of it.

3.4.1 Use case: Alpine

A pLasma Physics mINiapp for Exascale (ALPINE) is a set of mini-apps simulating well known problems in plasma physics using PIC, part of IPPL [6]. Since they all have well studied physical solutions, they are particularly well-suited for testing new features of IPPL. All mini-apps use the `FFTPeriodicPoissonSolver` class of IPPL. The mini-app simulating the phenomenon of weak Landau Damping [28, 29] was used as a candidate for testing the new mixed precision feature. Following the criterion described in the previous section, a study on the best type configuration for fields and particle attributes was carried, leading to the configuration of Table 3.3. The reason for it is that some tests showed that during the scatter phase, having charges in single precision would cause an error on the charges interpolated on the particles higher than machine precision for `float` (around 10^{-7}), which eventually saturated around 10^{-6}. As a consequence, ρ is set in double precision, while the particle position attribute and the electric field $\vec{E}$ were set in single precision (also justified with them being the data structures with the largest memory impact).

Table 3.3: Type configuration for mixed precision Landau Damping.

	`double`	`float`
Particle charge	✓	✗
Particle position	✗	✓
ρ field	✓	✗
$\vec{E}$ field	✗	✓

3.5 Kokkos Memory events

IPPL's back-end makes use of the performance portable library Kokkos [30] for its data structures. The library provides a high-performance programming model in C++, with features such as architecture-agnostic abstractions for both data structures and parallel regions, support for mixed execution spaces and fast copying from host to device. Other than the library itself, the Kokkos ecosystem also provides a variety of profiling and debugging tools which interface directly with hooks built inside the Kokkos enviroment at runtime [31]. The Kokkos tools are extremely user readable and can provide execution space specific or even kernel specific information. For all the memory studies in this work, the Kokkos memory profiling tool Memory Events was used. The tool keeps track of the total size, maximum memory detected per MPI rank (High water memory) and memory per timestep of every execution space declared in a Kokkos-based application at runtime.

3.6 Hardware

3.6.1 CPU

All of the CPU tests were performed on the Merlin cluster located at Paul Scherrer Institute (PSI). The cluster has a system of four chassis, and the ones used were the first three containing 25 nodes/chassis. Each node in chassis 0-2 has an Intel Xeon Gold 6152 processor, with 44 cores and 384GB of memory.

3.6.2 GPU

The GPU runs for IPPL required cuda 12, and gcc 11. For single-GPU tests (e.g. convergence and memory, see following chapter), an NVIDIA RTX 2080 Ti from the cluster gMerlin6 with 11GB of available memory was used. For scaling the solver, we used the Gwendolen node on the same partition, which has 8 NVIDIA A100s with Cuda v. 12 support and 40GB of memory. The heFFTe test runs were performed on the Leconte node from Innovative Computing Lab (ICL) at University of Tennessee (UTK), equipped with 8 NVIDIA V100 with 16GB memory each.

Chapter 4

Results and Discussion

4.1 Mixed precision

Before proceeding with the open boundary solver's result, memory and accuracy tests on type independency and mixed precision were carried. We ran the ALPINE mini-app for weak Landau damping with the configuration described in Table 3.3, and performed tests on accuracy and memory saving.

4.1.1 Accuracy studies

The configuration for this test was taken from [6] to reproduce the results from [29]. This means a grid size of 32^3, 8^6 particles, and time step size of 0.05 s (400 time steps in total). The resources used were 8 Merlin CPU nodes, with 16 MPI ranks in total (2 ranks per node). Figure 4.1 shows that the mixed precision run damps with a damping ratio close to the analytical one, as its double precision counter-part does. The difference plot reveals that the difference of the solution evaluated at each time step is between 10^{-11} and 10^{-7}. This result confirms what is expected from a mixed precision run, having an accuracy between the machine precision for `double` (10^{-14}) and the one for `float` (10^{-7}).

4.1.2 Memory studies

After checking the accuracy we procedeed with the memory test. With the same grid size and resource configuration as before, we started with a total of $32^3 \cdot 2560$ total particles, and increased the number up to $32^3 \cdot 5600$. The results detected on the host space by the Kokkos memory events tool are shown in Figure 4.2. With the mixed precision runs, we observe a consistent memory saving of 30% with respect to the double precision runs. Combining it with the results from Figure 4.1, it is shown that by enabling mixed precision in IPPL, the user is given the possibility of exploring configurations that can save memory without significantly deviating from the accuracy that 64-bits numbers can offer.

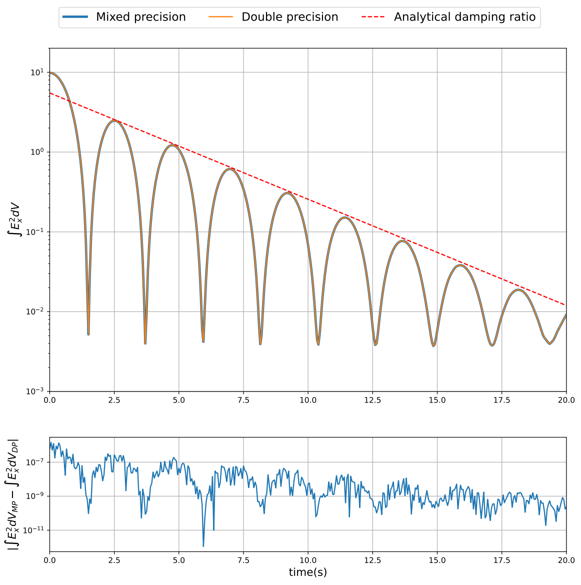


Figure 4.1: Weak Landau Damping of the electric field energy in the x-direction as a function of time, as in [6], with 32^3 grid points, 83, 886, 080 total particles and a time step size of 0.05 s. The difference plot of the energy between the mixed and double precision runs is shown at the bottom.

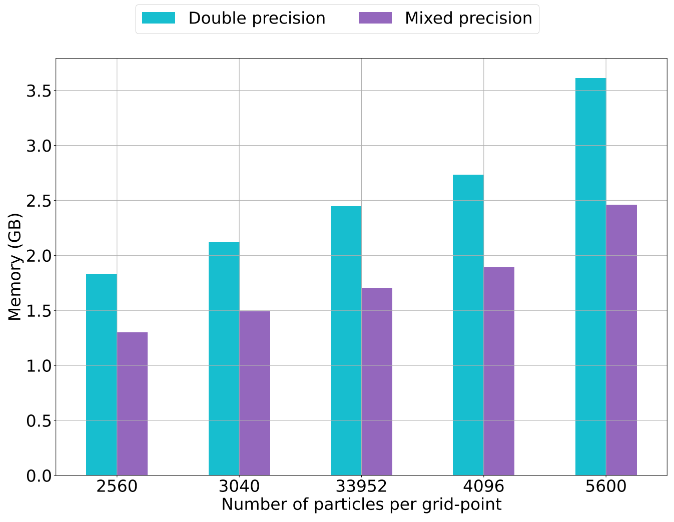


Figure 4.2: Weak Landau Damping mini-app memory study with 8 nodes, 2 MPI ranks/node and a grid size of 32^3, with number of particles/gridpoints progressively increasing.

4.2 Improved Vico Solver

4.2.1 Convergence studies

To check the correctness of the optimized solver with respect to the original, we compare the convergence results by running both versions with a smooth Gaussian source for both the potential and the electric field. The test is performed five times, decreasing the mesh spacing at each iteration. More specifically, the test program is ran with grid sizes of $4^3, 8^3, 16^3, 32^3, 64^3, 128^3$. The execution is repeated on both CPU and GPU.

As shown in Figure 4.3, the errors measured for the improved version agrees with the original ones. This happens for both the scalar field and the vector field. There are slight differences for finer grids, which are negligible since both numbers are below machine precision. With this test, we have proven that the optimized algorithm is, in fact, correct.

4.2.2 Memory studies

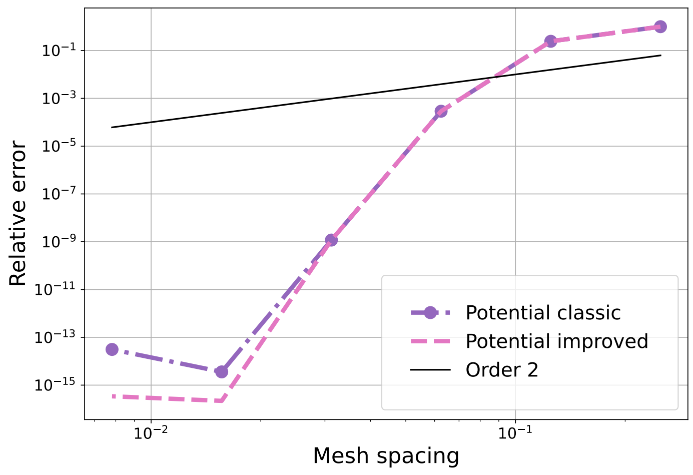
After verifying the convergence of the optimized Vico solver, we procedeed to quantify the memory improvement we get from the algorithmic change. This is essential in order to determine the largest fitting problem sizes for weak and strong scaling studies.

The same test with the Gaussian source used for the scaling has been repeated for this case. The CPU test was executed on a single CPU node on Merlin, and for the GPU a was requested. The choice of the problem size was done in the following way. The problem sizes chosen for the study are $32^3, 64^3, 96^3, 128^3, 256^3$, using the theoretical back-of-the-envelope calculations of Table 4.1 as an estimator for the maximum fitting problem. For the GPU study, the runs stop at 128^3 because of the reduced memory availability of a single GPU. The information about the memory usage is retrieved from the host space tracking in the CPU runs, and from the Cuda space tracking in the GPU runs.

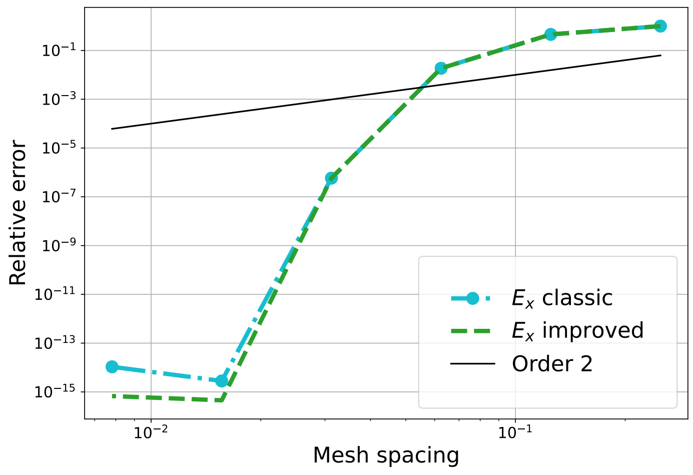
Figure 4.4 shows the astonishing memory saving that was obtained with the algorithmic improvement. Both memory sizes for CPU and GPU agree as expected, and the results are comparable with the theoretical ones of Table 4.1, thus their correctness is confirmed. The total memory occupied by the improved Vico solver is around 50% and 70% of the one used by the original. Moreover, an extra problem size can be fit on both devices with the improved algorithm, before both versions of the solver exceed maximum memory. This will be extremely important for the runs on multiple nodes, as we can run the scaling analysis on larger problem sizes.

	Vico	Improved Vico
32^3	$\sim 10^2$ MB	$\sim 5 \times 10^1$ MB
64^3	$\sim 5 \times 10^2$ MB	$\sim 10^2$ MB
128^3	$\sim 5 \times 10^3$ MB	$\sim 10^3$ MB

Table 4.1: Theoretical estimation for the total memory (in MB) needed to store the fields' data structures in the Solver class, for original and improved Vico.



a. Convergence test for potential field.



b. Convergence test for electric field.

Figure 4.3: Convergence tests for classic and improved Vico. The current state-of-the art convergence, which is second order, is also depicted for comparison.

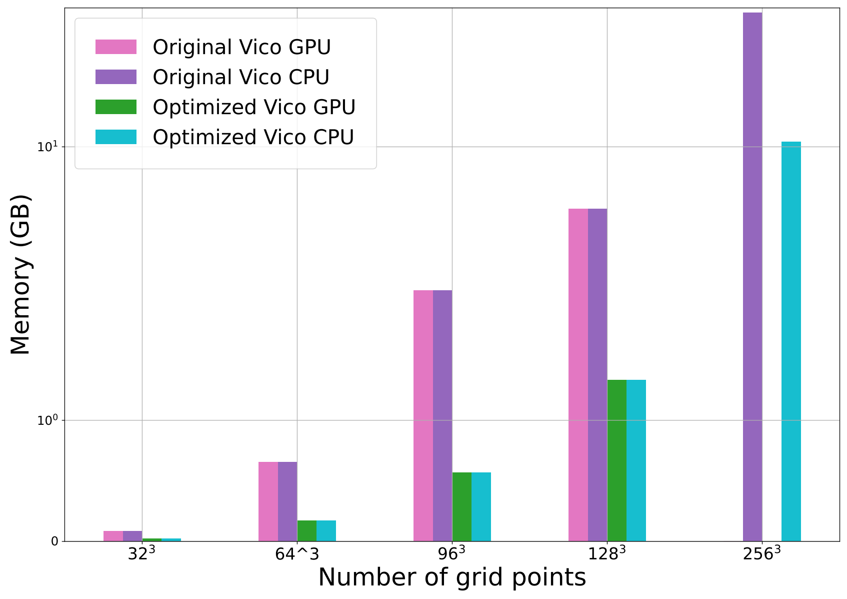


Figure 4.4: Maximum memory tracked by Kokkos MemEvents over a run of the Gaussian test, increasing grid points number up to the largest size before memory overflow.

4.2.3 Scaling studies

All the tests in this section were repeated for single and double precision for both CPUs and GPUs. The candidate test program is the same used for the memory studies, with every test being repeated five times per run in order to reduce noise in the timings. We benchmarked the new, improved version of the Vico solver, and compared the timings curve, the efficiency and the speedup against the ones already observed for the vanilla Vico version [1]. Runs with all combinations of the different heFFTe parameters for rank decompositions and communication were tested to determine the better ones for both solvers. All of the tests are conducted on PSI's Merlin cluster (see section 3).

Strong scaling

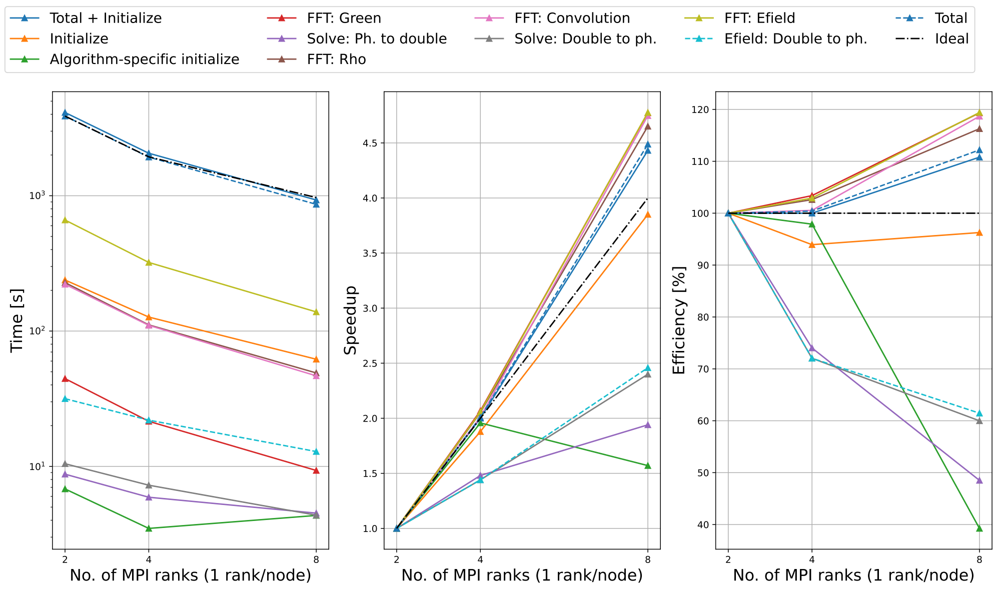
We procedeed by deciding the minimum number of CPU nodes/GPUs, and we selected the problem size accordingly, choosing the largest fitting one for the improved Vico solver. On CPU, this number corresponds to 512^3 for the improved Vico solver and 256^3 for the vanilla Vico solver. We then increased the number of nodes up until 8, using 1 MPI rank per node and 44 OMP threads, allowing us to exploit all of the CPUs on the node. The results showed in Figure 4.5 and Figure 4.6 show the results for the best configuration of heFFTe parameters, however, not much difference was observed on CPU for different combinations of parameters (see Appendix). Significant changes with precision weren't observed either. In the runs with the improved version of Vico, the difference between the actual total runtime without initialization and the one detected by the main timer is more subtle since the extra initialization time is given by the extension of the domain, which is of course smaller for this case. We can conclude that we have achieved the same scaling for both solvers with the same level of speedup and efficiency, very close to the ideal curves and independently on the parameters and precision.

For GPU runs, the largest fitting size in one of the NVIDIA A100 GPUs of Gwendolen was 256^3 for the improved Vico solver, and 128^3 for the vanilla Vico solver. We ran the scaling on the Gwendolen node using from 2 up to the maximum number of GPUs (8). The test program ran with 1 MPI rank/GPU. From Figure 4.7 we observe that the efficiency stays around 90% without going below for both double and single precision, and a speedup slightly above 3.5 (very close to the ideal speedup of 4) is obtained. Comparing it to Figure 4.8, it is immediate to see that on GPU the vanilla solver performs slightly worse for single precision (Figure 4.8a) than double precision (see Figure 4.8b), probably due to communication offset. However, even for the better case of the double precision run, both efficiency (close to 80%) and speedup (< 3.5) do not overcome the ones in Figure 4.7b.
Even in the worst case configurations, the improved solver obtained better performance and scalability than the vanilla version (see Appendix 5). We can then conclude that the algorithmic improvement does what it was meant for in the first place: solver applications can now bear larger problem sizes, overcoming communication offsets and thus achieving better results than the previous version.

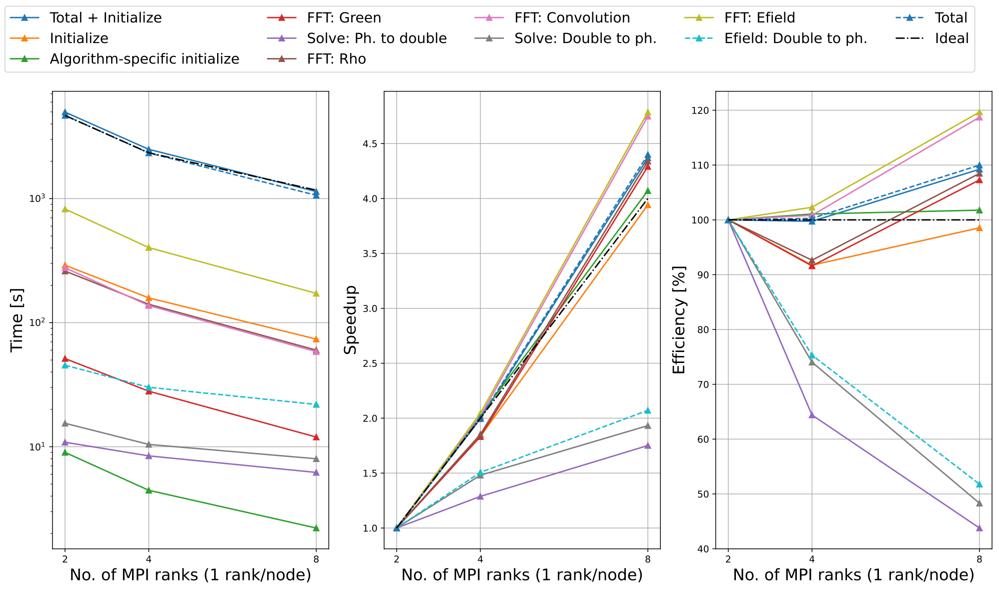
Weak scaling

For CPU, we started with a 256^3 grid size for 2 CPU nodes in the case of the improved version (which turns into 128^3 for the vanilla version). We increased progressively and proportionally until 8 nodes. The number of MPI ranks and OMP threads used, as well as the program ran,

[1]Note that the results for the old version of the solver differ from the ones shown in [3]. The difference comes from the update of heFFTe to version 2.3, released in winter 2022, one year and a half after the study was carried.

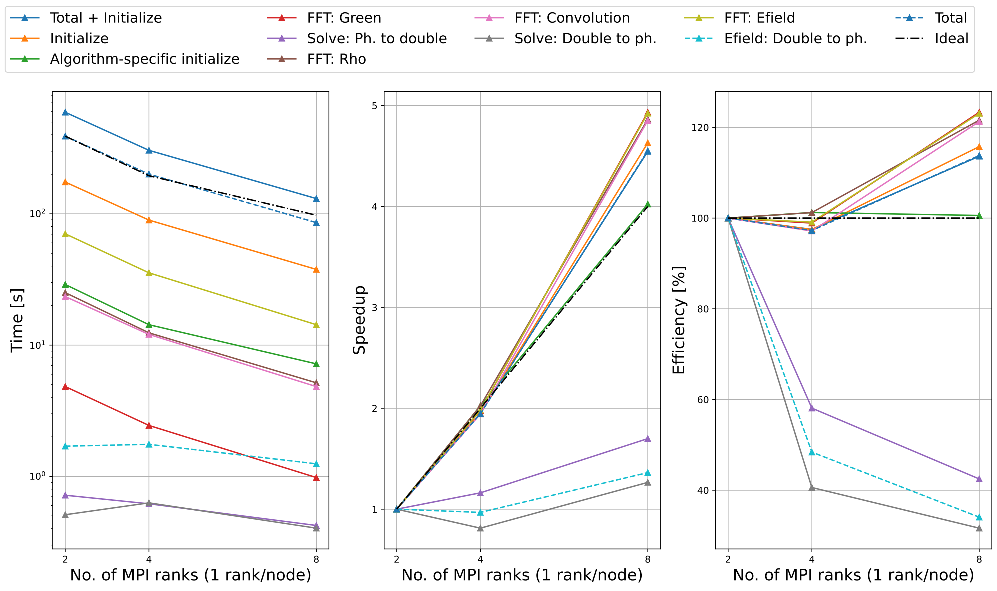


a. Single precision run with slabs decomposition, all-to-all communication and reordering.

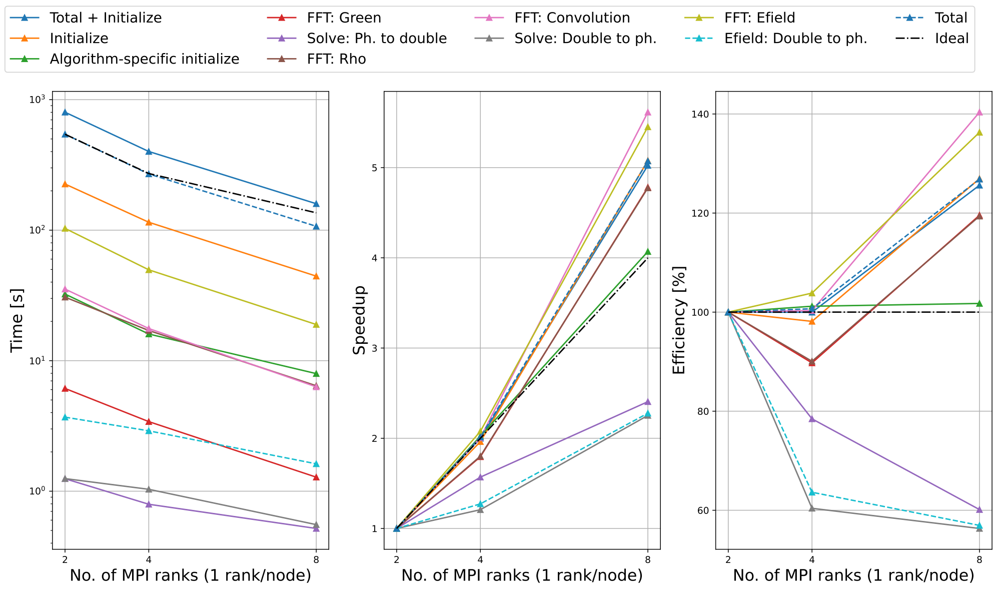


b. Double precision run with slabs decomposition, pipelined point-to-point communication and reordering.

Figure 4.5: Strong scaling results on CPU for five iteration of solving the Poisson equation with a Gaussian source, using the Improved Vico solver. The best heFFTe parameters are shown for the two runs. Total grid points are $N_x N_y N_z = 512^3$

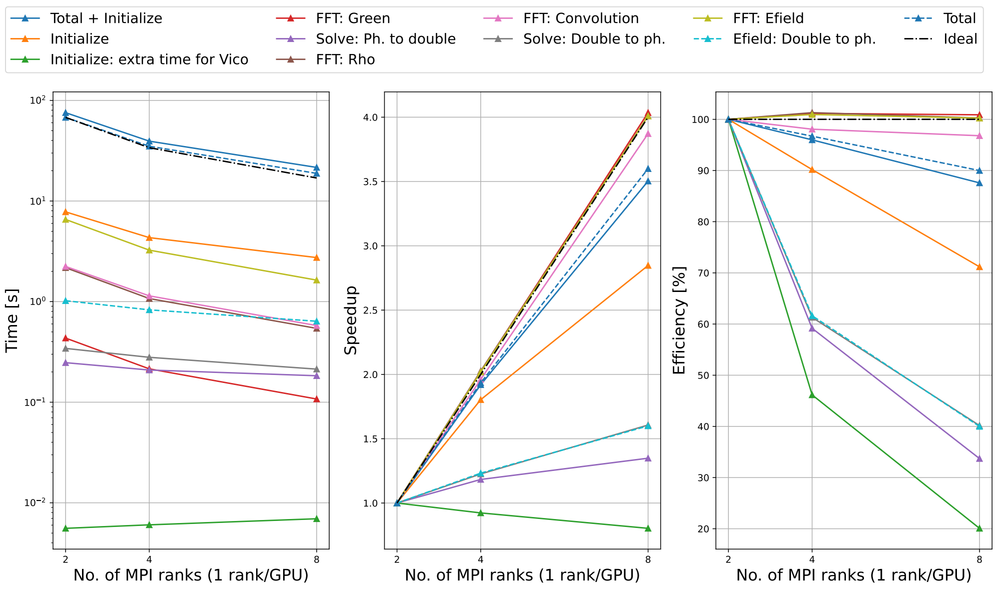


a. Single precision run with slabs decomposition, all-to-all-v communication and reordering.

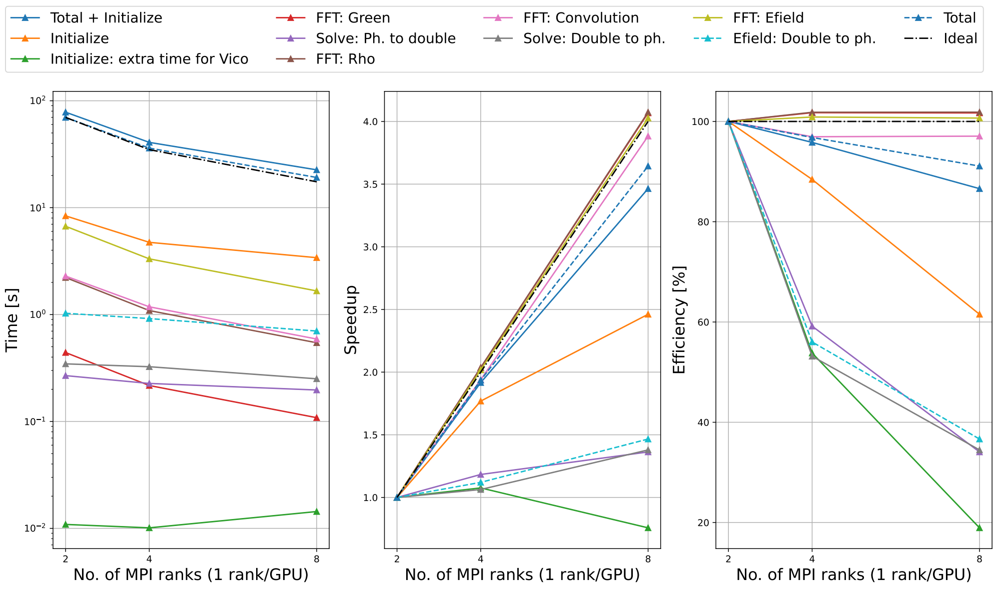


b. Double precision run with slabs decomposition, pipelined point-to-point communication and reordering.

Figure 4.6: Strong scaling results on CPU for five iteration of solving the Poisson equation with a Gaussian source, using the vanilla Vico solver. The best heFFTe parameters are shown for the two runs. Total grid points are $N_xN_yN_z = 256^3$

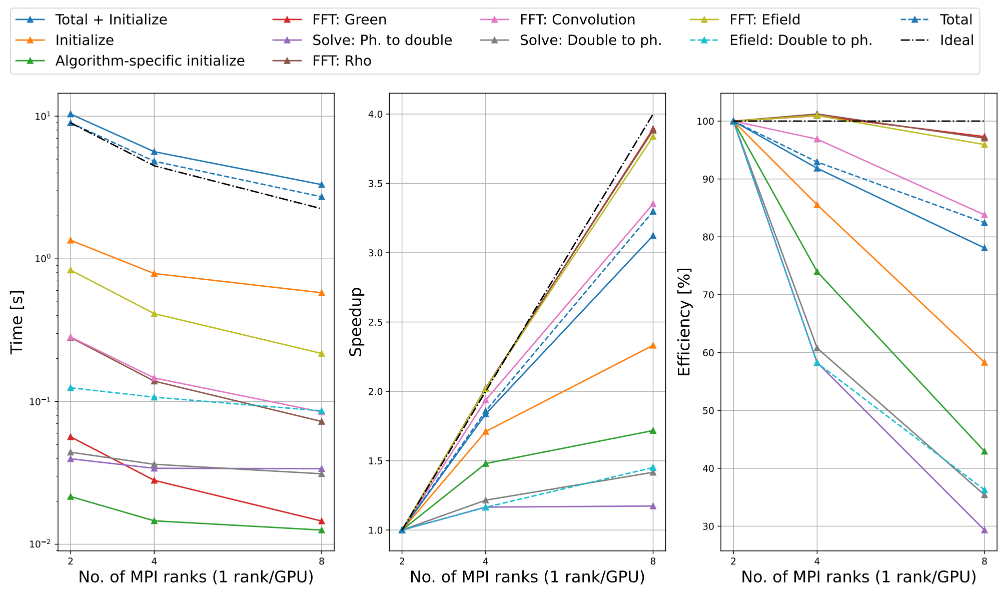


a. Single precision run with pencils decomposition, point-to-point communication and reordering.

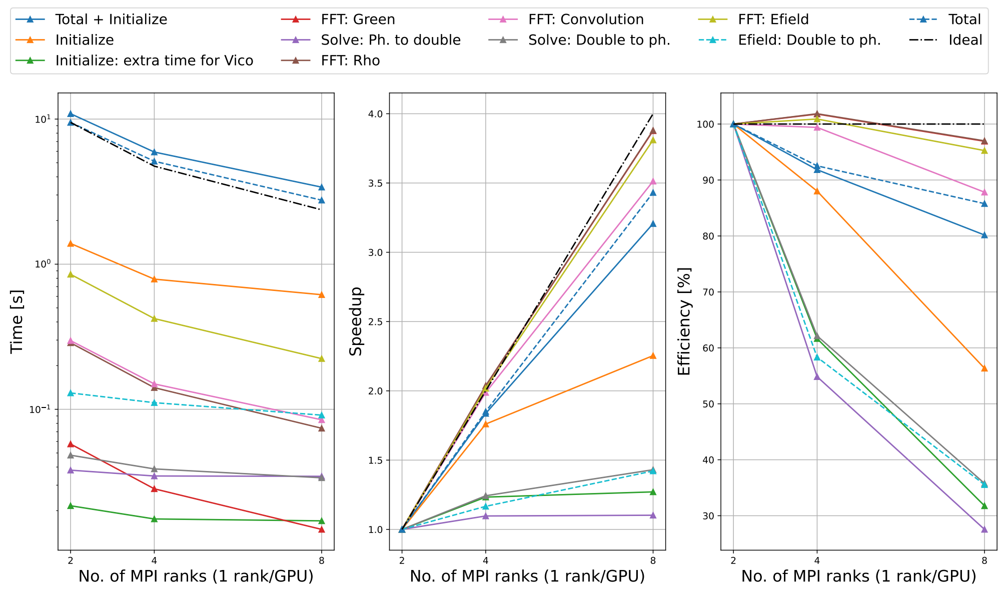


b. Double precision run with slabs decomposition, pipelined point-to-point communication and reordering.

Figure 4.7: Strong scaling results on GPU for five iteration of solving the Poisson equation with a Gaussian source, using the improved Vico solver. The best heFFTe parameters are shown for the two runs. Total grid points are $N_xN_yN_z = 256^3$.



a. Single precision run with pencils decomposition, all-to-all-v communication and reordering.



b. Double precision run with slabs decomposition, all-to-all-v communication and reordering.

Figure 4.8: Strong scaling results on GPU for five iteration of solving the Poisson equation with a Gaussian source, using the vanilla Vico solver. The heFFTe parameters selected are: pencil decomposition, pipelined point-to-point communication, without reordering. Total grid points are $N_x N_y N_z = 128^3$.

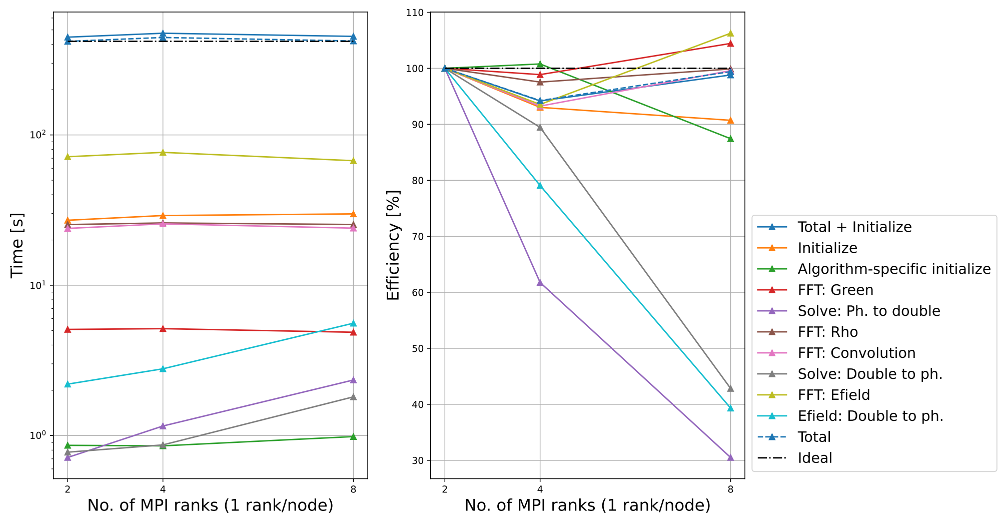
is the same as in the strong scaling study. Using slabs instead of pencils turned out to be the better choice, and the results shown in Figure 4.9 are the ones with this parameter set. The type of communication did not seem to affect the runs. From Figure 4.9 it can be observed that both the single and double precision runs do not flutcuate away from the ideal scaling more than within a 2% range, while the efficiency obtained is around 90% in the best case. The double precision run (Figure 4.9b) had in general a slightly higher efficiency in alll the runs than the single precision one (Figure 4.9b). Similar results can be observed also for Figure 4.10, but the differences between the two precisions was always less evident. he important observation remains that also for this case, the new solver scales roughly like the original.

On GPUs, we selected 256^3 to be the problem size for 2 GPUs, and increased proportionally, keeping the workload per GPU constant, up to 8 GPUs. For the vanilla solver, we kept the configuration from [3], which started from 128^3 for two GPUs and increased accordingly. For the improved solver, the different heFFTe parameters configuration did not differ significantly from one another, independently from the run's precision, giving results almost identical to the ones shown in Figure 4.11. None of the run went below the the 90% limit and time didn't rise in a significant way, staying close to the ideal horizontal line. This was not the case for the vanilla solver, where some parameter choices, especially in the double precision case, were less optimal. But in the best case, Figure 4.12 shows that weak scaling is achieved and the efficiency does not deviate too much from the horizontal line, staying around 90%. Here as well we observe similar results as the ones encountered for the strong scaling: if we compare Figure 4.11 and Figure 4.12, we see that weak scaling is achieved with both solvers and the efficiency still lays close to the ideal horizontal line. This is a further demonstration that with a problem size one power of two higher, the effects of communication are less visible and we are exploiting better the GPUs, leading to results that are equal or better than the ones previously achieved.

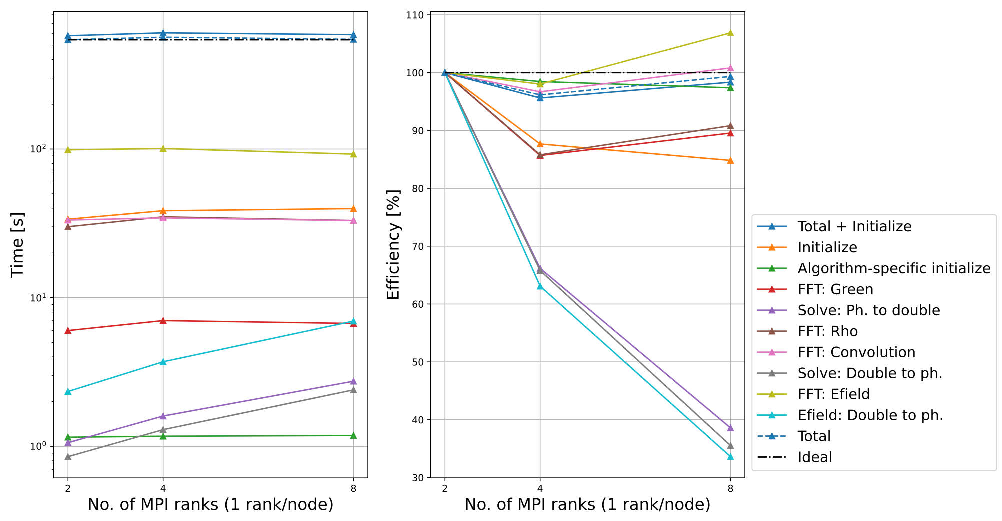
4.3 heFFTe Scaling studies

As an additional study, we compared the DCT-I computed with the native transform from the FFTW back-end with the non-native transform relying on R2C cuFFT transforms. For this purpose, we chose a fixed size of 512^3 for the strong scaling of both FFTW and CUFFT, and ran a full-cycle transform (backward and forward) 8 times, increasing MPI ranks from 2 to 16 (for CPU nodes) and from 2 to 8 (for GPUs). All the tests were carried in single precision on the Merlin cluster.

Figure 4.13 and Figure 4.14 show the scaling results, in terms of performance and timing, respectively with the FFTW and cuFFT backend. While the choice over pencils or slabs did not make a difference for FFTW, on the GPU runs with cuFFT it is evident from Figure 4.14a and Figure 4.14b that some configurations are more suboptimal than others. However, heFFTe's cuFFT DCT-I does not scale in any of the cases. This is probably just limited to the number of GPUS < 8. heFFTe FFTs on GPUs are hard to scale in general, due to communication limitations. However, for this case increasing the number of GPUs would not automatically imply better performance. If all GPUs share the same bandwidth, for example, more devices will not improve communication and the performance will not increase because of latency limits due to increased bandwidth.

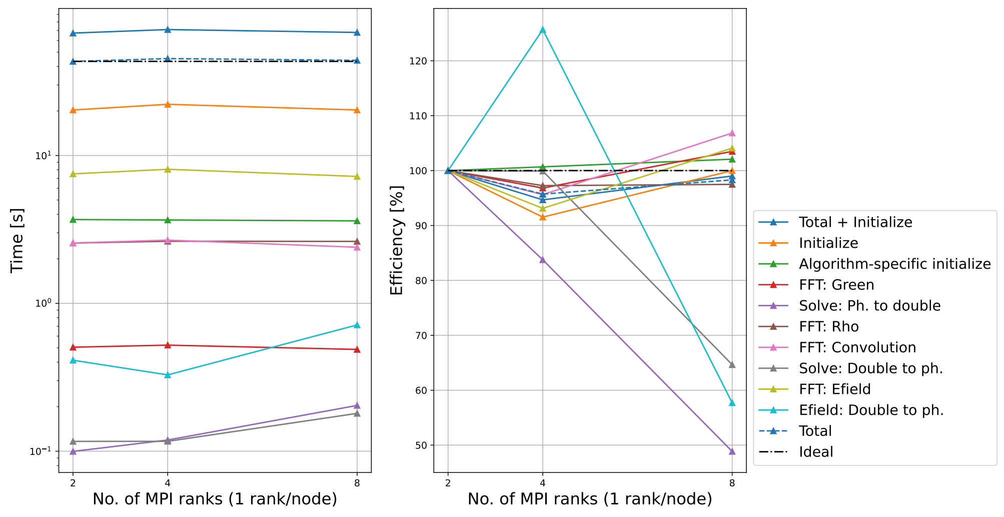


a. Single precision run with slabs decomposition, point-to-point communication and reordering.

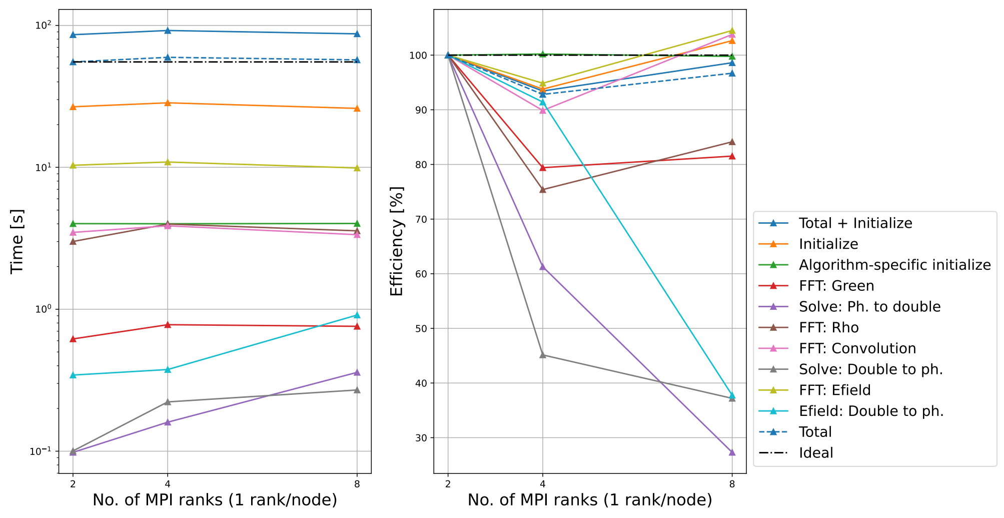


b. Double precision run with slabs decomposition, point-to-point communication and reordering.

Figure 4.9: Weak scaling results on CPU for five iteration of solving the Poisson equation with a Gaussian source, using the Improved Vico solver. The best heFFTe parameters are shown for the two runs. Grid size for smallest number of CPU nodes is $N_xN_yN_z = 256^3$.

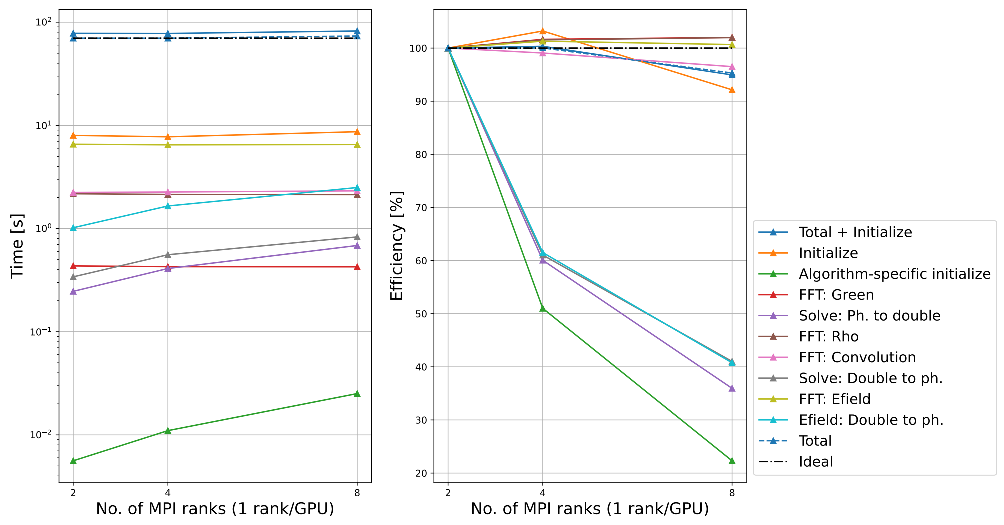


a. Single precision run with slabs decomposition, point-to-point communication and reordering.

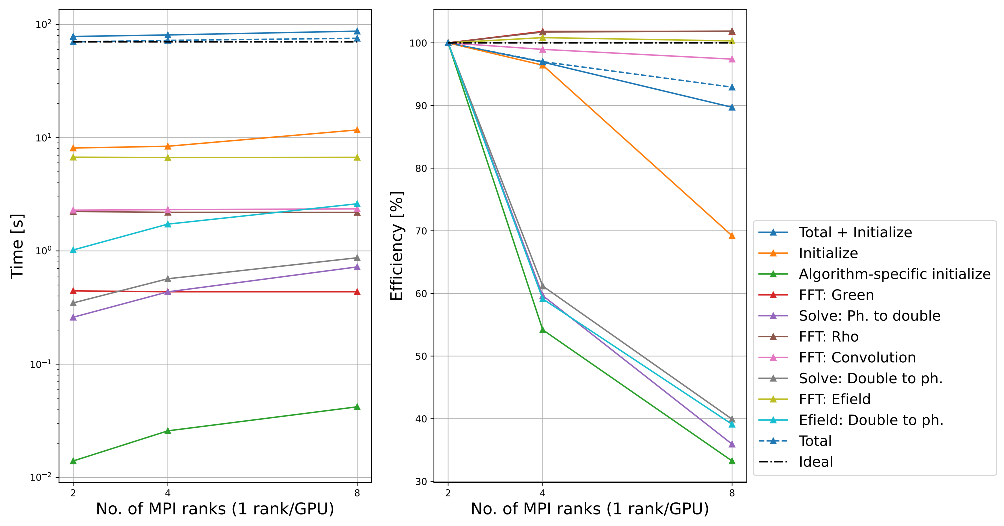


b. Double precision run with slabs decomposition, point-to-point communication and reordering.

Figure 4.10: Weak scaling results on CPU for five iteration of solving the Poisson equation with a Gaussian source, using the vanilla Vico solver. The best heFFTe parameters are shown for the two runs. Grid size for smallest number of CPU nodes is $N_xN_yN_z = 128^3$.

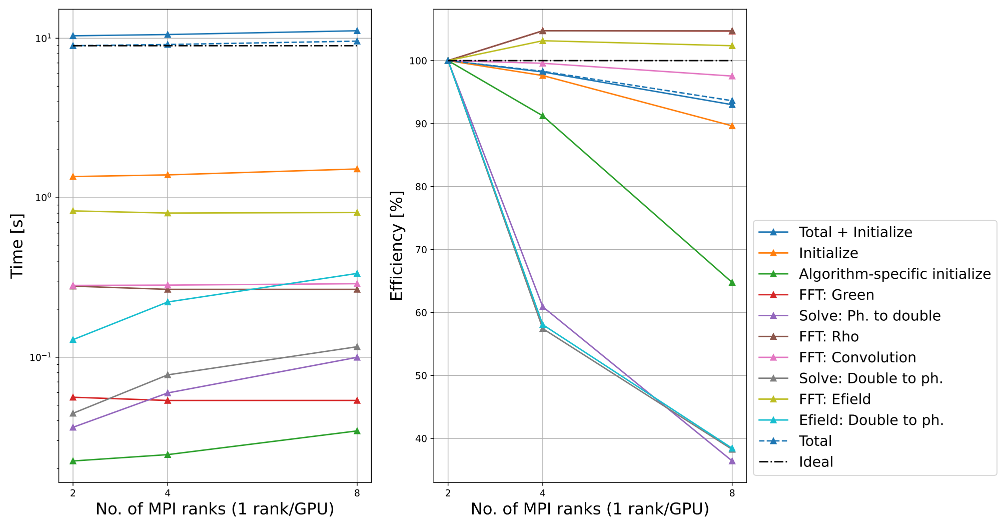


a. Single precision run with pencils decomposition, point-to-point communication and reordering.

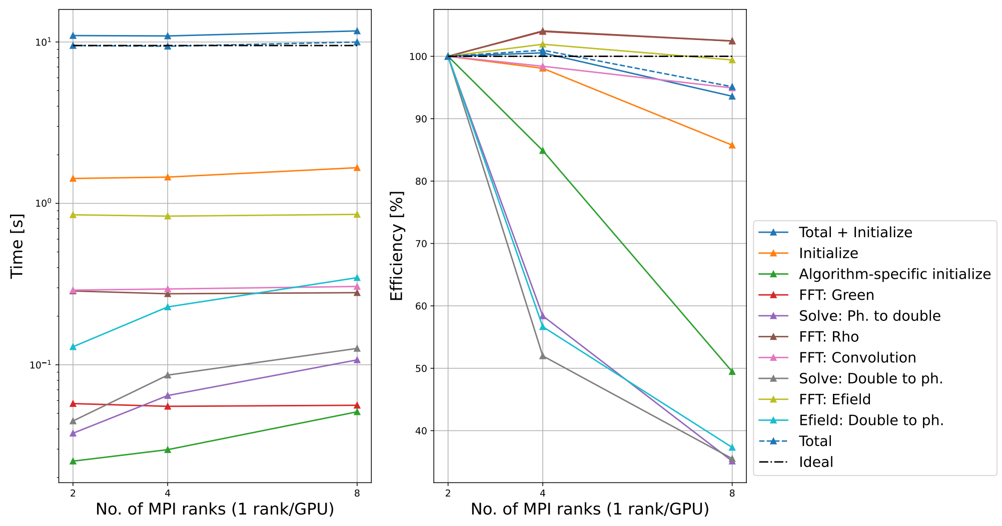


b. Double precision run with slabs decomposition, point-to-point communication and reordering.

Figure 4.11: Weak scaling results on GPU for five iteration of solving the Poisson equation with a Gaussian source, using the improved Vico solver. The best heFFTe parameters are shown for the two runs. Grid size for smallest number of GPUs is $N_x N_y N_z = 256^3$.



a. Single precision run with pencils decomposition, pipelined point-to-point communication and reordering.



b. Double precision run with slabs decomposition, point-to-point communication and reordering.

Figure 4.12: Weak scaling results on GPU for five iteration of solving the Poisson equation with a Gaussian source, using the vanilla Vico solver. The best heFFTe parameters are shown for the two runs. Grid size for smallest number of GPUs is $N_x N_y N_z = 128^3$.

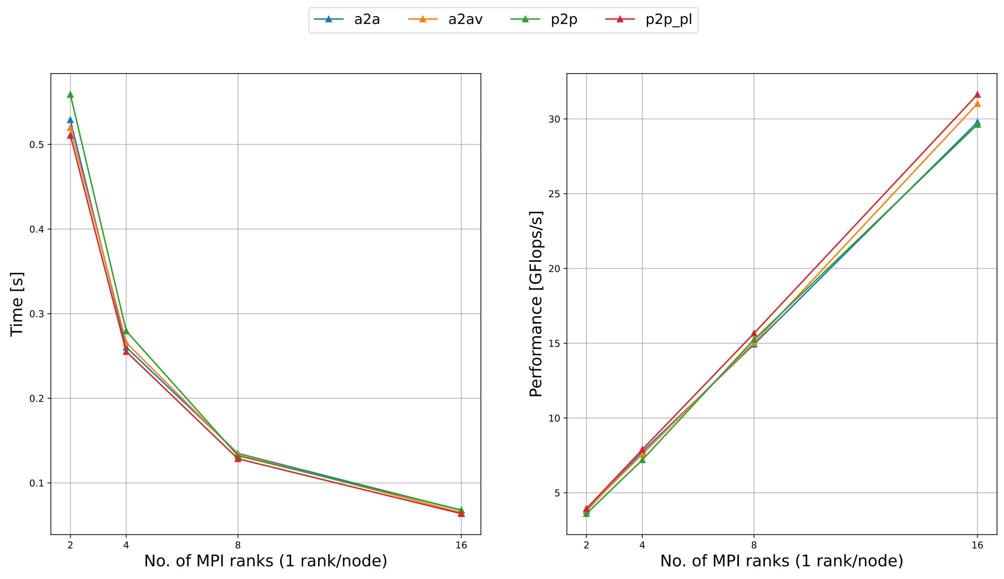
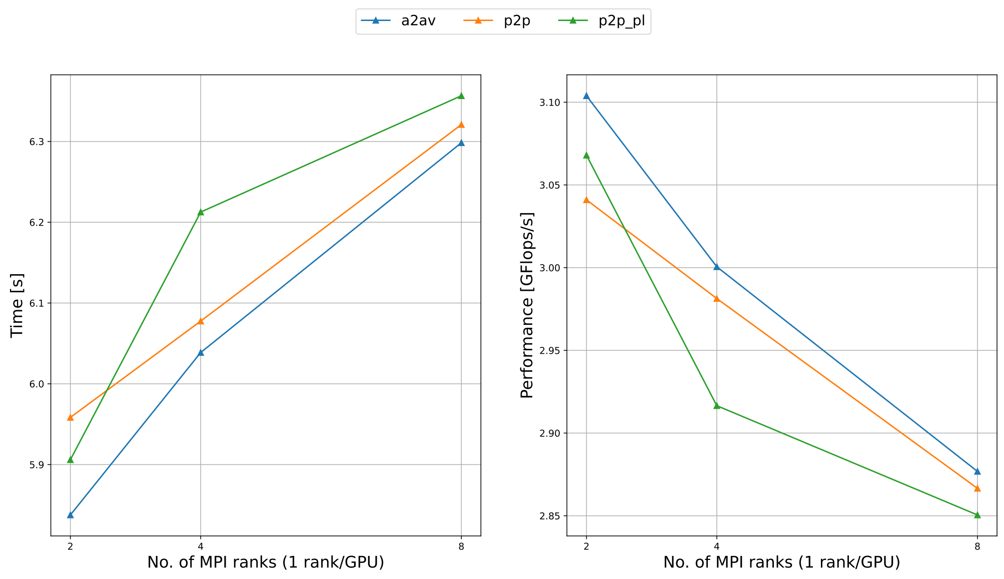
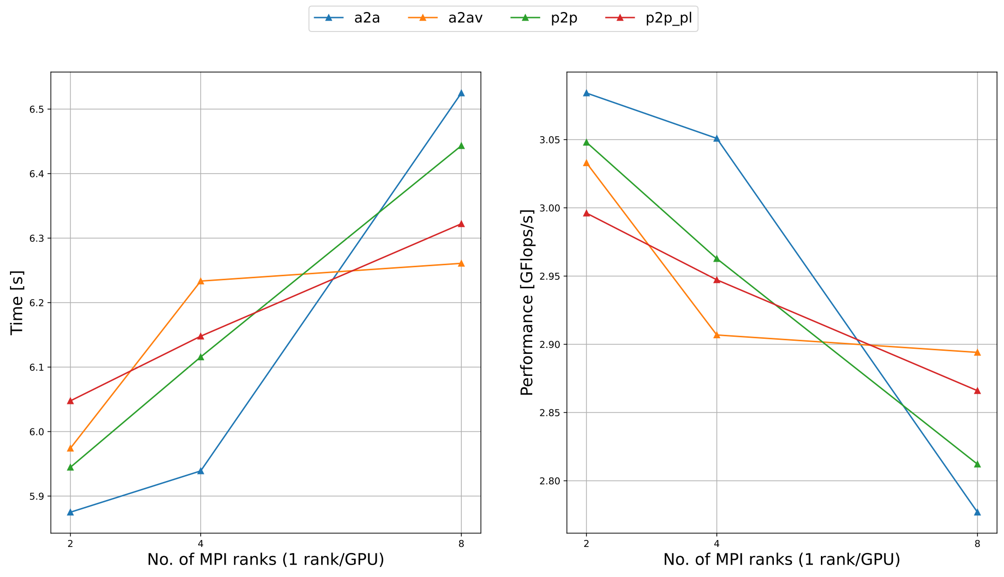


Figure 4.13: Strong scaling study for the DCT-I with the native FFTW backend, with pencils decomposition and for every communication option in heFFTe.



a. Pencils decomposition.



b. Slabs decomposition.

Figure 4.14: Strong scaling study for the DCT-I with the non-native cuFFT back-end, for every communication option in heFFTe.

Chapter 5

Conclusions and future work

Once the heFFTe kernels were implemented, the algorithmic substitution was immediate to include in IPPL, since it only required to substitute the R2C DFT with a DCT-I, and the $(4N)^3$ grid with a $(2N+1)^3$ one. For such an easy change, the achievement was an impressive memory cut down, spacing from 30% up to 50% of the one occupied by the vanilla implementation. This is necessary and extremely important for the GPUs runs. In fact, if previously two GPUss with the largest memory capacity on our cluster failed to fit more than 128^3 grid points, now they can handle without any problems runs with 256^3 grid points. In this context, managing to fit an extra power of two means having the possibility of studying performance more efficiently and increasing the workload per GPUs makes us capable of fully use all the power that they can offer. We have also observed an efficiency of $\sim 90\%$ in all the scalings, and in all architectures (both CPU and GPU) tested so far. The conclusion to be drawn at the end of this work is that now the Vico solver is competitive with the state-of-the art Hockney, as it uses similar memory occupation and similar timings, but with the much better accuracy from the regularization of G.

As an additional advantage, using heFFTe as a backbone for the FFT computation gives us access to communication and computation options, which always allows for optimal configurations for the best results possible, on every architecture.

Since the results on a small set of available resources are already promising, it would be interesting to observe the solver's behaviour on larger systems. Runs on machines like Oak Ridge National Lab's Frontier or NERSC's Perlmutter are already in program. As a further next step, the Vico solver would be a good candidate in accelerator modelling for elongated beams simulations.

On the heFFTe side, fixing of the normalization constants mismatches for all the R2R transforms flavours is in progress.

Bibliography

[1] F. Vico, L. Greengard, and M. Ferrando, “Fast convolution with free-space green’s functions,” *Journal of Computational Physics*, vol. 323, pp. 191–203, 2016.

[2] R. W. Hockney and J. W. Eastwood, “Computer simulation using particles,” 1966.

[3] S. Mayani, “A Performance Portable Poisson Solver for the Hose Instability,” 2021.

[4] A. Adelmann, P. Calvo, M. Frey, A. Gsell, U. Locans, C. Metzger-Kraus, N. Neveu, C. Rogers, S. Russell, S. Sheehy, J. Snuverink, and D. Winklehner, “OPAL a Versatile Tool for Charged Particle Accelerator Simulations,” 2019.

[5] A. Ayala, S. Tomov, A. Haidar, and J. Dongarra, “heFFTe: Highly Efficient FFT for Exascale,” in *International Conference on Computational Science (ICCS 2020)*, (Amsterdam, Netherlands), 2020-06 2020.

[6] S. Muralikrishnan, M. Frey, A. Vinciguerra, M. Ligotino, A. J. Cerfon, M. Stoyanov, R. Gayatri, and A. Adelmann, “Scaling and performance portability of the particle-in-cell scheme for plasma physics applications through mini-apps targeting exascale architectures,” 2022.

[7] V. Britanak, P. C. Yip, and K. Rao, “Chapter 1 - discrete cosine and sine transforms,” in *Discrete Cosine and Sine Transforms* (V. Britanak, P. C. Yip, and K. Rao, eds.), pp. 1–15, Oxford: Academic Press, 2007.

[8] “CUDA Nvidia: cuFFT Library.” https://docs.nvidia.com/cuda/cufft/. Date accessed: 2023.09.12.

[9] W. J. Dally, S. W. Keckler, and D. B. Kirk, “Evolution of the Graphics Processing Unit (GPU),” *IEEE Micro*, vol. 41, no. 6, pp. 42–51, 2021.

[10] S. Pennycook, J. Sewall, and V. Lee, “Implications of a metric for performance portability,” *Future Generation Computer Systems*, vol. 92, pp. 947–958, 2019.

[11] P. L. Pritchett, *Particle-in-Cell Simulation of Plasmas— A Tutorial*, pp. 1–24. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003.

[12] B. Alder, S. Fernbach, M. Rotenberg, and J. Gillis, “Methods in Computational Physics,” *Physics Today*, vol. 17, pp. 48–50, 08 1964.

[13] J. Shen, T. Tang, and L.-L. Wang, *Spectral Methods: Algorithms, Analysis and Applications.* Berlin, Heidelberg: Springer Berlin Heidelberg, 2011.

[14] R. D. Ryne, “On FFT-based convolutions and correlations, with application to solving Poisson’s equation in an open rectangular pipe,” 2011.

[15] 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".

[16] E. Brigham and R. Morrow, "The fast Fourier transform," *IEEE Spectrum*, vol. 4, no. 12, p. 63 – 70, 1967. Cited by: 244.

[17] J. W. Cooley and J. W. Tukey, "An algorithm for the machine calculation of complex fourier series," *Mathematics of Computation*, vol. 19, no. 90, p. 249 – 259, 1965. Cited by: 6782; All Open Access, Bronze Open Access.

[18] J. Eastwood and D. Brownrigg, "Remarks on the solution of poisson's equation for isolated systems," *Journal of Computational Physics*, vol. 32, no. 1, pp. 24–38, 1979.

[19] J. T. Rasmussen and J. H. Walther, "Particle methods in bluff body aerodynamics," 2011.

[20] J. Zou, E. Kim, and A. J. Cerfon, "FFT-based free space Poisson solvers: why Vico-Greengard-Ferrando should replace Hockney-Eastwood," 2021.

[21] N. Ahmed, T. Natarajan, and K. Rao, "Discrete Cosine Transform," *IEEE Transactions on Computers*, vol. C-23, no. 1, pp. 90–93, 1974.

[22] V. Britanak, P. C. Yip, and K. Rao, "Chapter 2 - definitions and general properties," in *Discrete Cosine and Sine Transforms* (V. Britanak, P. C. Yip, and K. Rao, eds.), pp. 16–50, Oxford: Academic Press, 2007.

[23] "FFTW official website." http://www.fftw.org/. Date accessed: 2023.09.12.

[24] "Intel® oneAPI Math Kernel Library." https://www.intel.com/content/www/us/en/developer/tools/oneapi/onemkl.html. Date accessed: 2023.09.12.

[25] "Matlab Discrete cosine transform documentation." https://www.mathworks.com/help/signal/ref/dct.html. Date accessed: 2023.09.12.

[26] "Scipy.fft Discrete cosine transform documentation." https://docs.scipy.org/doc/scipy/tutorial/fft.html#type-ii-dct. Date accessed: 2023.09.12.

[27] "FFTW documentation." http://www.fftw.org/fftw3_doc/index.html#SEC_Contents. Date accessed: 2023.09.12.

[28] L. F. Ricketson and A. J. Cerfon, "Sparse grid techniques for particle-in-cell schemes," *Plasma Physics and Controlled Fusion*, vol. 59, p. 024002, dec 2016.

[29] A. Ho, I. A. M. Datta, and U. Shumlak, "Physics-based-adaptive plasma model for high-fidelity numerical simulations," *Frontiers in Physics*, vol. 6, 2018.

[30] C. R. Trott, D. Lebrun-Grandié, D. Arndt, J. Ciesko, V. Dang, N. Ellingwood, R. Gayatri, E. Harvey, D. S. Hollman, D. Ibanez, N. Liber, J. Madsen, J. Miles, D. Poliakoff, A. Powell, S. Rajamanickam, M. Simberg, D. Sunderland, B. Turcksin, and J. Wilke, "Kokkos 3: Programming model extensions for the exascale era," *IEEE Transactions on Parallel and Distributed Systems*, vol. 33, no. 4, pp. 805–817, 2022.

[31] C. Trott, L. Berger-Vergiat, D. Poliakoff, S. Rajamanickam, D. Lebrun-Grandie, J. Madsen, N. Al Awar, M. Gligoric, G. Shipman, and G. Womeldorff, "The Kokkos EcoSystem: Comprehensive Performance Portability for High Performance Computing," *Computing in Science Engineering*, vol. 23, no. 5, pp. 10–18, 2021.

Acknowledgements

First, I would like to thank the person that made this work and my whole year at PSI possible, Dr. Andreas Adelmann. I could not have asked for a better supervisor for both my semester project and my thesis. From the choice of the topic, through the semester and up until the hand-in, he always made sure that I was engaged and motivated in the work, and gave me extremely useful professional advice.
A big thanks to Sonali Mayani, she was also an amazing supervisor, always taking her time to help in case there were problems. I am particularly glad to have worked side by side with her in the United States.
Thanks to the IPPL group, in particular to Dr. Sriramkrishnan Muralikrishnan and Dr. Matthias Frey to always be available and to show me around IPPL. Thanks to Alessandro Vinciguerra for the insights and tips on `C++`.
Thanks also to the rest of the AMAS group members, Arnau, Renato, Mathieu, Raphael, Sichen, Tobia, Sebastian, Bastien and Guglielmo for making the experience at PSI lovely.

Thanks to the Linear Algebra group at ICL for making my two months at UTK possible, and thanks to Hartwig Anzt for welcoming me in it.
Especially, i would like to thanks Prof. Stan Tomov and Dr. Miroslav Stonyatov for supervising me during my stay, for guiding me through the project and expressing interest in it. I enjoyed particularly the `C++` hackatons, where I have learned a lot and shared my previous knowledge.
Thanks also to Theresa Finchun and Tracy Lee for helping me with immigration and for always making sure that everything was going smoothly.

Thanks to Prof. Marco Scalerandi and Prof. Amelia Sparavigna from Politecnico di Torino, for furtherly igniting my interest in physics and writing my reccomendation letters for ETH, without them all of this would not have been possible.

Thanks to my family and friends back in Italy, which love never faded despite the distance. The whole of Sambucone Molinari, Giacomo, Francesco, my amazing DnD party and my Dungeon Master Leonardo, and the former sixth floor from Campus San Paolo.
A thanks goes also to the Mu Nu Chapter of IEEE Eta Kappa Nu, especially to the other members of the AA 2021/22 board, for making me part of a vibrant and active community and giving me the best extra-universitary experience.
Thanks to Reto, Lea, Lukas, Lucius, Cheng, Miray, Roman, Jessica, Seva, Edgars, Tianxu, Liine, Todd, Julius, Daniel, Alvaro and Juner, the special people I have met here that accompanied me and supported me, sharing all the joys and pains of the Master's years.

Appendix

Appendix A: Code

The IPPL framework can be found on GitHub at the address https://github.com/IPPL-framework[1]. For the code of the improved Vico solver, check branch number 121. heFFTe's master branch already contains the changes made during this work. The cuda kernels for the DCT-I are implemented in `src/heffte_backend_cuda.cu`. For more details on the work's timeline check commits 6814400, 86f51a5 and 5c2b95b.

[1]GitLab is no longer available.

Appendix B: Memory estimation

The calculations to estimate the memory occupied by both solvers were done by summing all the field sizes in function of the grid size in the n-th direction N (the fields are listed in Table 5.1), counting an additional factor of two for the complex fields. The final result in GB is obtained by substituting N and multiplying the final result by 8 (which is the size in bytes for double-precision numbers), except for the only integer field. Moreover, the test program `TestGaussian.cpp` has two additional fields of size N^3 for the exact solution of ρ and E.

Table 5.1: Fields present in the `FFTPoissonSolver` class for both Vico and improved Vico.

Field name	Meaning	Size	Type
`rho`	Charge density ρ	N^3	Real
`E`	Electric field	$3(N^3)$	Real
`storage_field`	Reference for all fields using the doubled grid	$(2N)^3$	Real
`rho_tr`	FFT-transformed ρ on doubled grid	$(2N)^3$	Complex
`grn_tr`	FFT-transformed Green's fct. restricted on doubled grid	$(2N)^3$	Complex
`grnL`	FFT-transformed Green's fct. on quadrupled grid (Vico only)	$(4N)^3$	Complex
`grn_2n1`	dct-transformed Green's fct on halved grid (improved Vico only)	$(2N+1)^3$	Complex
`temp`	Temporary storage for solution ϕ	$(2N)^3$	Real
`grnI`	Field facilititating calculation of Green's fct.	$3(2N)^3$	Integer

Appendix C: extra CPU scaling results

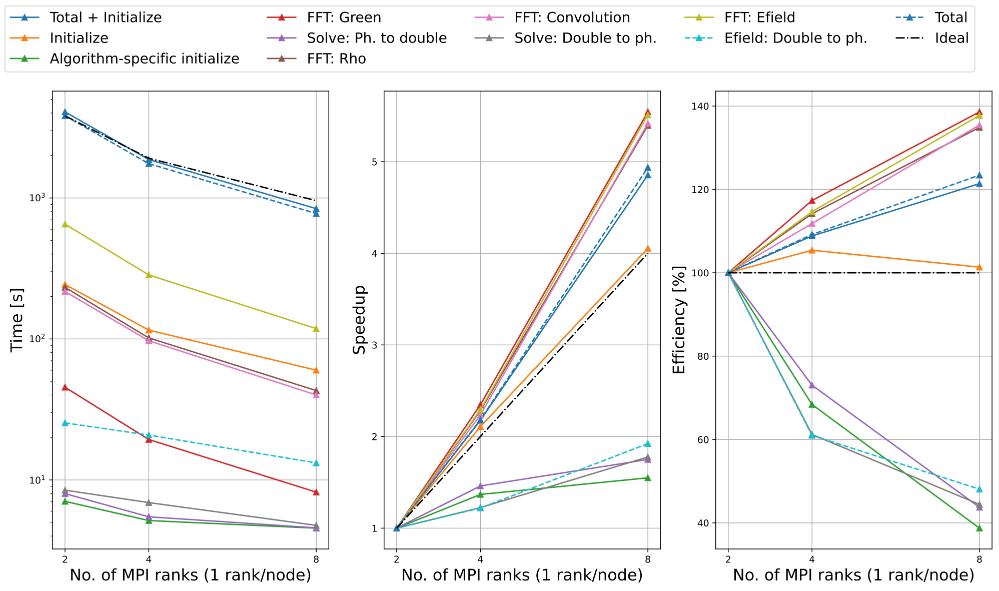


Figure 5.1: Strong scaling in single precision for improved Vico, with pencils decomposition, all-to-all communication and reordering.

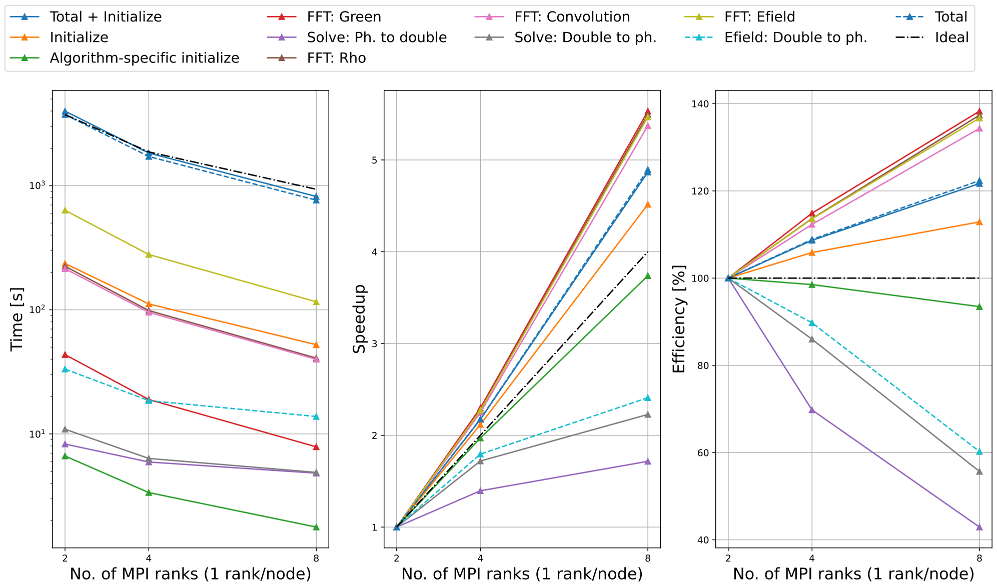


Figure 5.2: Strong scaling in single precision for improved Vico, with pencils decomposition, pipelined point-to-point communication and reordering.

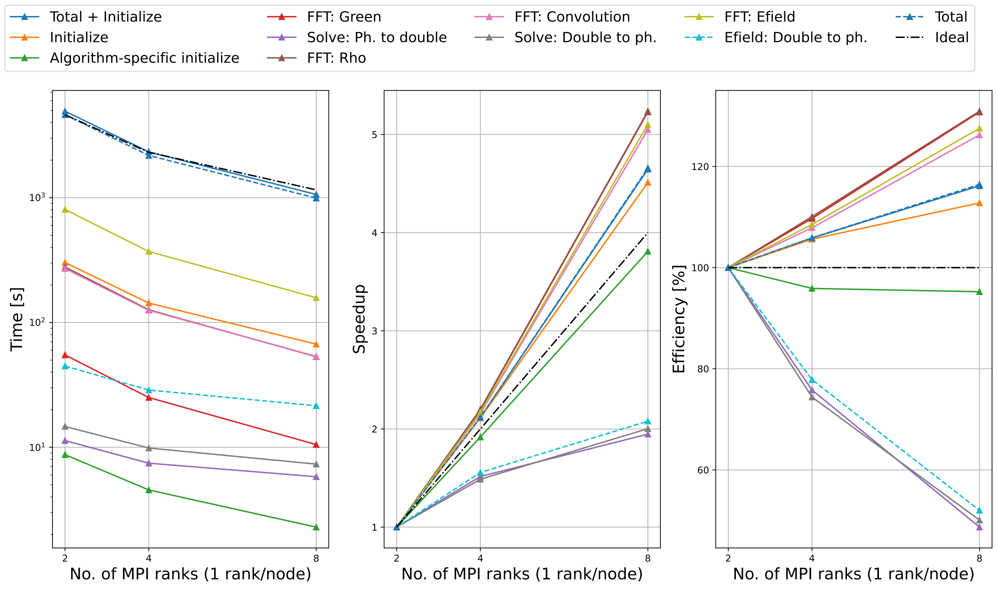


Figure 5.3: Strong scaling in double precision for improved Vico, with pencils decomposition, all-to-all-v communication and reordering.

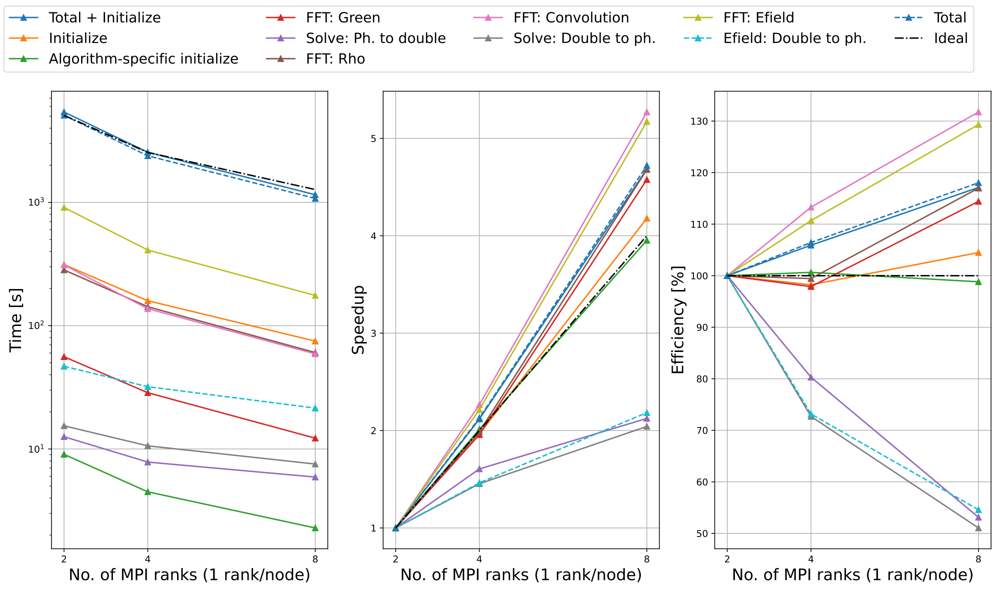


Figure 5.4: Strong scaling in double precision for improved Vico, with slabs decomposition, all-to-all-v communication and reordering.

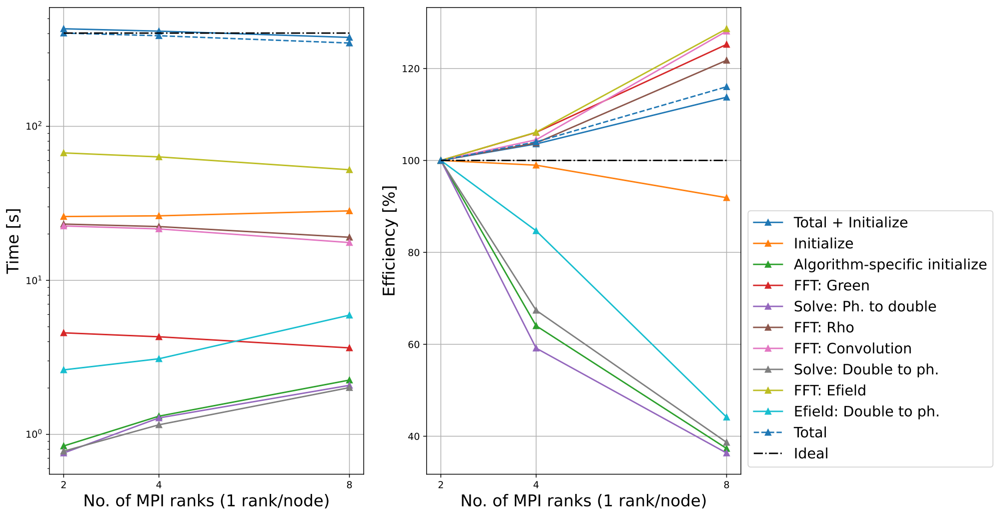


Figure 5.5: Weak scaling in single precision for improved Vico, with pencils decomposition, all-to-all communication and reordering.

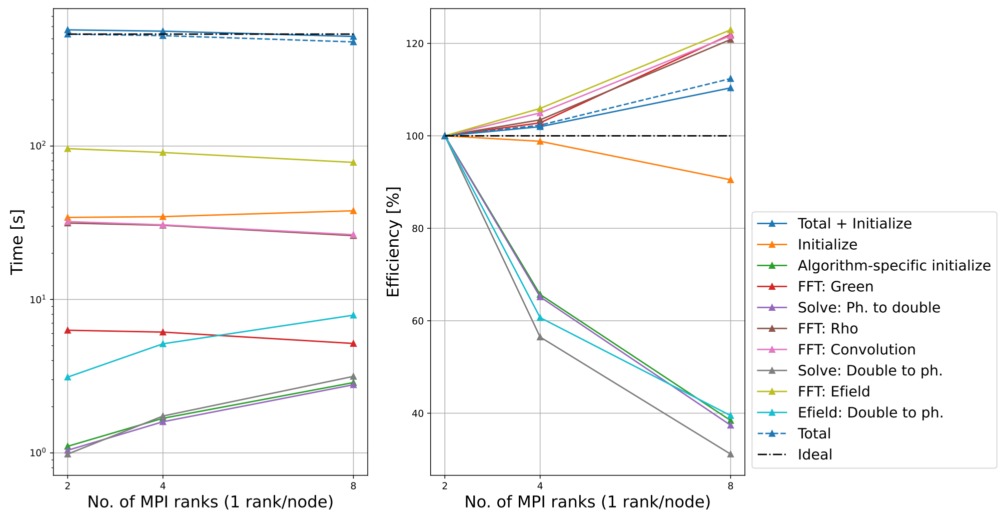


Figure 5.6: Weak scaling in single precision for improved Vico, with slabs decomposition, all-to-all communication and reordering.

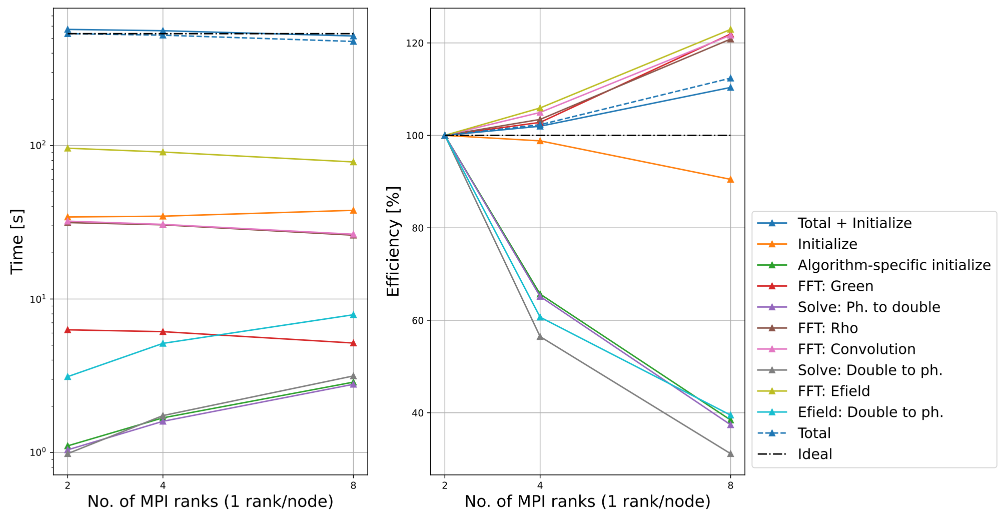


Figure 5.7: Weak scaling in double precision for improved Vico, with pencils decomposition, all-to-all communication and reordering.

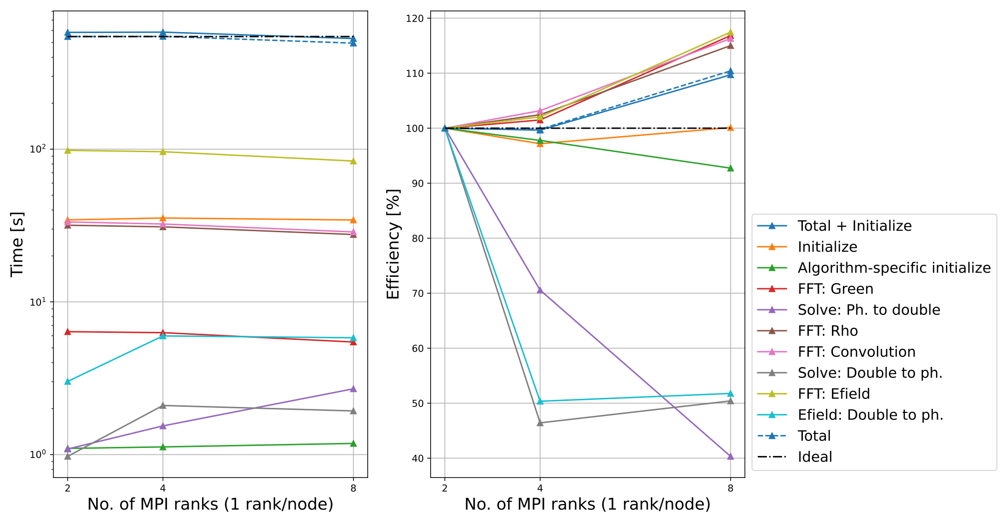


Figure 5.8: Weak scaling in double precision for improved Vico, with pencils decomposition, point-to-point communication and reordering.

Appendix D: extra GPU scaling results

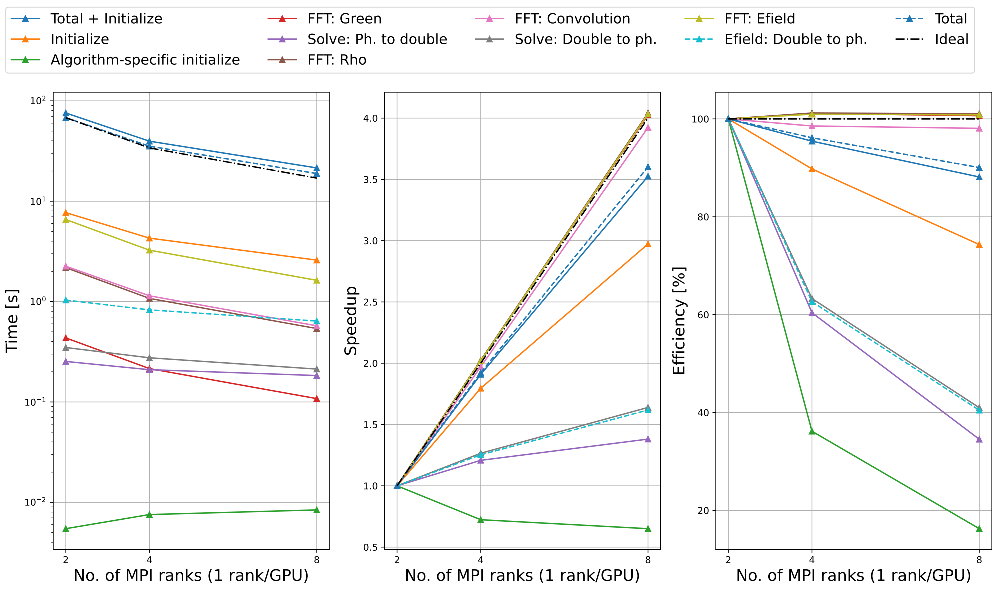


Figure 5.9: Strong scaling in single precision for improved Vico, with pencils decomposition, all-to-all-v communication and reordering.

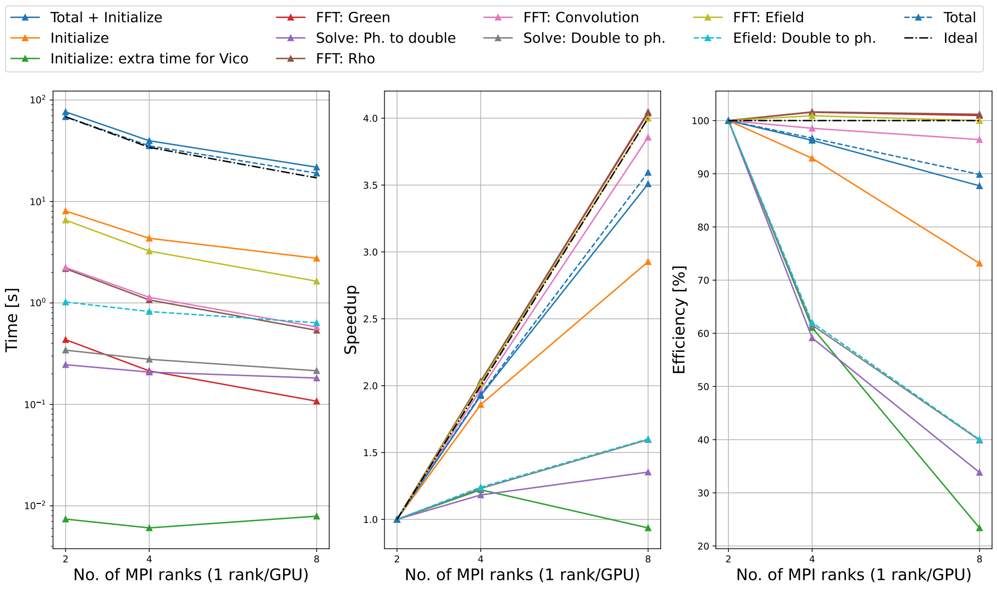


Figure 5.10: Strong scaling in single precision for improved Vico, with slabs decomposition, point-to-point communication and reordering.

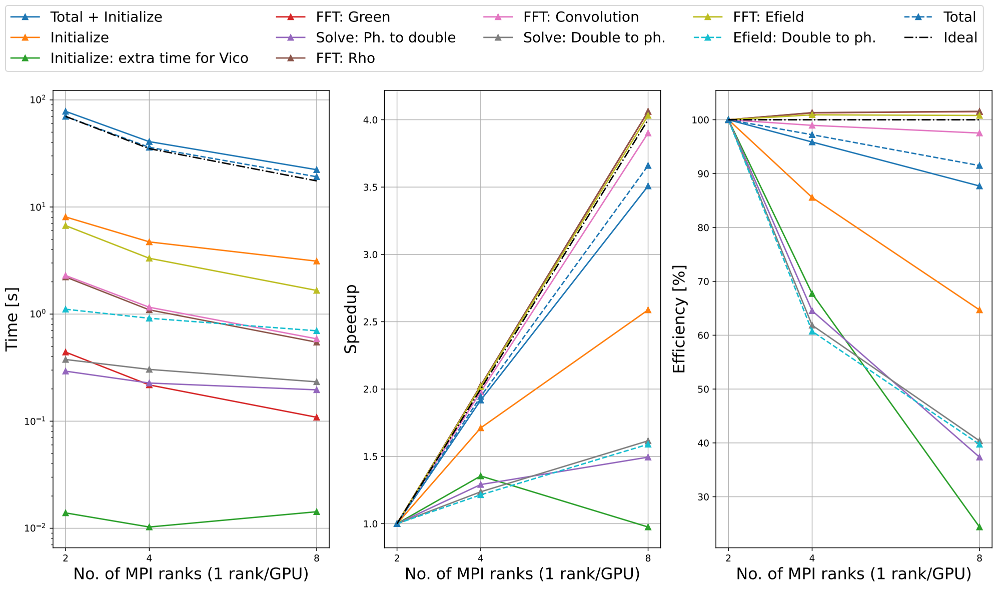


Figure 5.11: Strong scaling in double precision for improved Vico, with pencils decomposition, pipelined point-to-point communication and reordering.

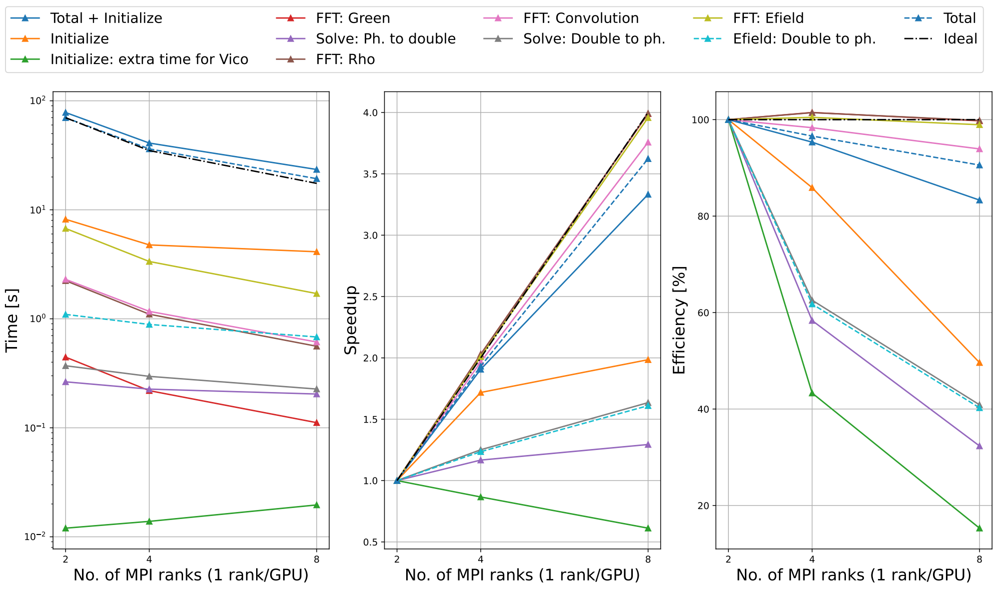


Figure 5.12: Strong scaling in double precision for improved Vico, with slabs decomposition, all-to-all communication and reordering.

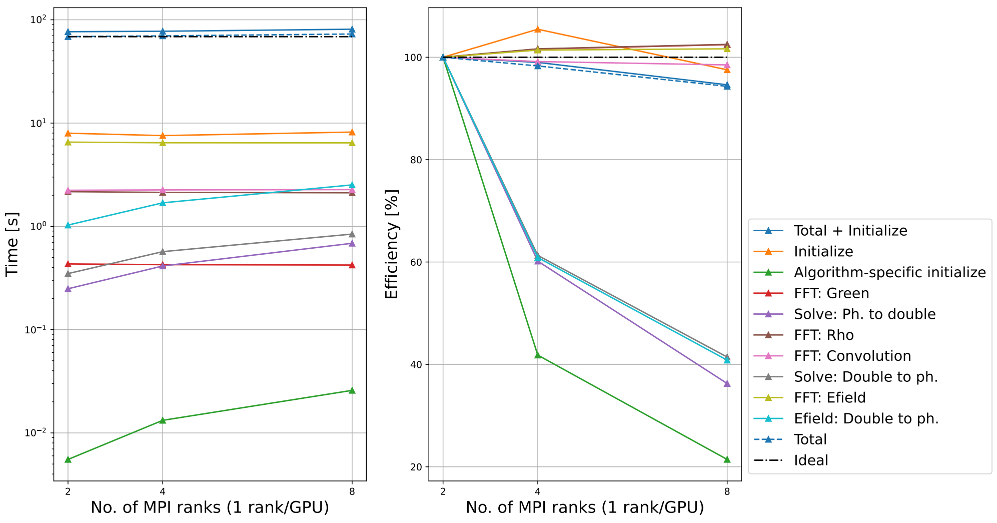


Figure 5.13: Weak scaling in single precision for improved Vico, with slabs decomposition, pipelined point-to-point communication and reordering.

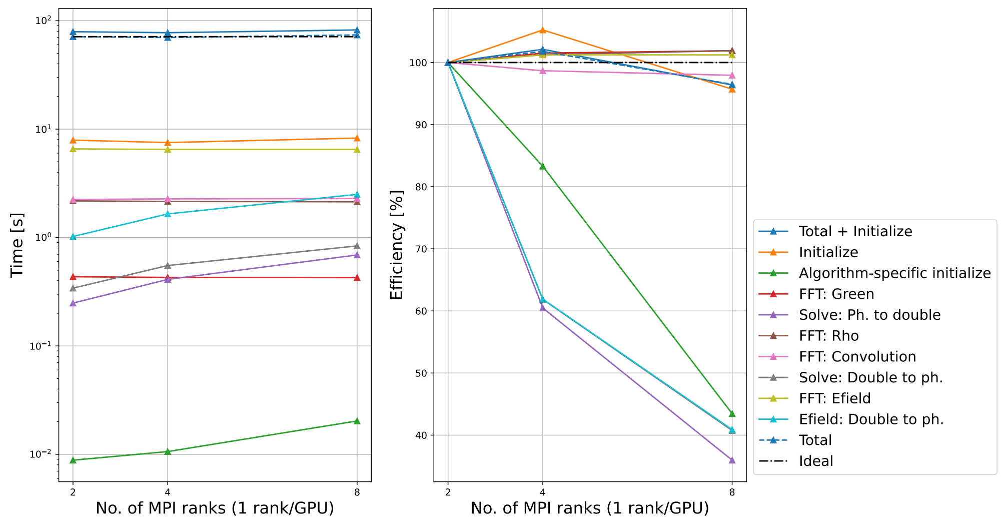


Figure 5.14: Weak scaling in single precision for improved Vico, with pencils decomposition, all-to-all-v communication and reordering.

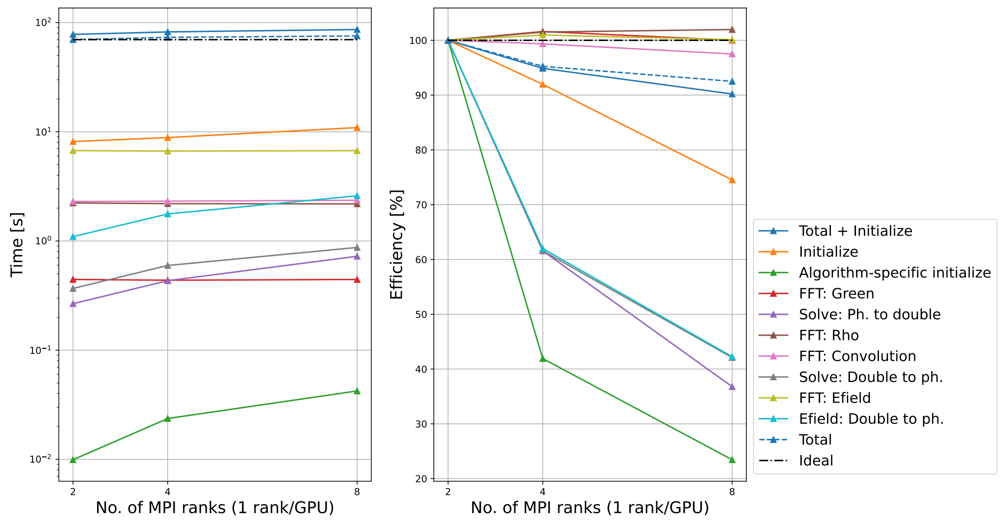


Figure 5.15: Weak scaling in double precision for improved Vico, with pencils decomposition, point-to-point communication and reordering.

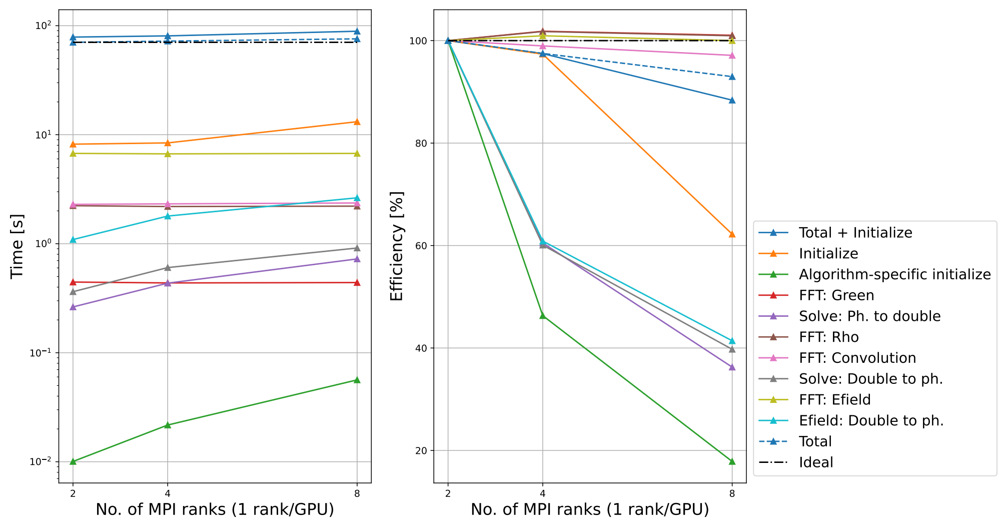


Figure 5.16: Weak scaling in double precision for improved Vico, with slabs decomposition, all-to-all communication and reordering.