A Performance Portable Poisson Solver for the Hose Instability

Master Thesis

Master in Physics, EPFL

written by

Sonali Mayani

supervised by

Dr. A. Adelmann (PSI)
Prof. A. Pautz (EPFL)

scientific advisors

Prof. A. Cerfon (NYU)
Dr. M. Frey (University of St. Andrews)
Dr. S. Muralikrishnan (PSI)

August 25, 2021

Abstract

High-performance computing is an impressive tool for physical simulations that allows us to move towards more accurate and reliable computational models, as it cuts down simulations times significantly. As computing architectures become increasingly complex and potent, there is a push towards exascale computing (10^{18} operations per second). Exascale computing architectures are heterogeneous: they consist of a combination of processor types. To take full advantage of these heterogeneous architectures, computational simulations need to be efficient and hardware portable, *i.e.* performance portable. The aim of this thesis is to precisely create such performance portable code, focusing on Particle-in-Cell (PIC) methods, which are used in various fields such as plasma physics, astrophysics, and accelerator physics. Concretely, a Poisson solver for open boundary conditions is created in the framework of the Independent Parallel Particle Layer (IPPL), a backend of the accelerator simulation tool OPAL [1]. We make use of the Kokkos ecosystem [2], a tool produced by the Exascale Computing Project in order to allow for portable code. The Poisson solver will be based on Fast Fourier Transforms (FFTs), provided by the HeFFTe library [3], whose aim is to make FFT scalable and available for exascale computing. Finally, the Poisson solver's performance is shown via scaling studies.

Contents

List of Figures

Acronyms

ALPINE A pLasma Physics mINiapp for Exascale

CPU Central Processing Unit

ECP Exascale Computing Project

FFT Fast Fourier Transform

GPU Graphics Processing Unit

HeFFTe Highly Efficient FFT for Exascale

HPC High-Performance Computing

IPPL Independent Parallel Particle Layer

MPI Message Passing Interface

OPAL Object-oriented Parallel Accelerator Library

PIC Particle-in-Cell

Chapter 1

Introduction

During the past half a century, there has been a consistent increase in performance of computer processors, as shown by Moore's law. This can mostly be attributed to the progress of the semiconductor industry, which has vastly invested in making smaller transistors, allowing to fit more of them in integrated circuits (Figure 1.1). However, this steady improvement of microprocessors is not guaranteed in the future. The trend is slowing down as transistors approach the quantum scale, since new effects such as quantum tunneling come into play [4].

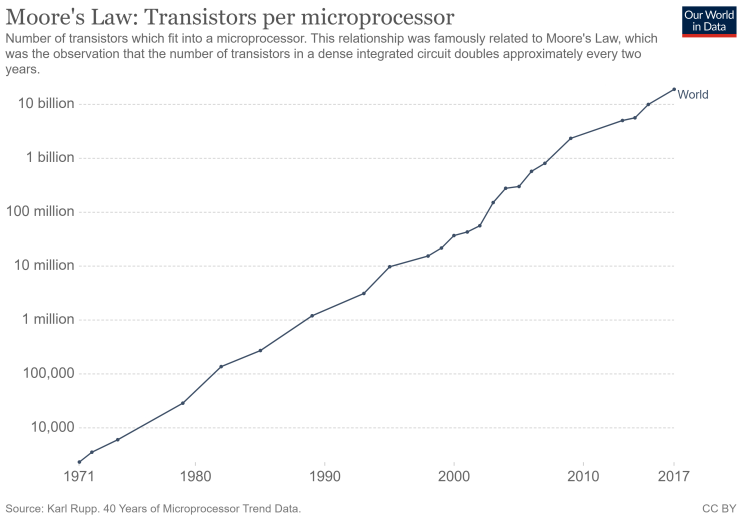

Figure 1.1: Evolution of the number of transistors per microprocessor since 1971 [5].

Currently, the limiting factor for performance increase of single-processors is the power consumption. When processors are performing operations, they dissipate heat as they use electric current, and so require cooling systems. If processors were doing more operations per second than what is currently achievable, the heat dissipated would be so large that no existing cooling system would be able to deal with it; cooling technology has progressed slower than processor technology [6, chap. 3]. Hence, making faster single-processors is not a feasible endeavour.

Parallelization on homogeneous architectures (with a single type of processor) allowed to increase performance without the need for faster processors up to the petascale (10^{15} operations per second). However, homogeneous architectures are not sufficient if one wants to go beyond petascale computing. High-Performance Computing (HPC) is delving into alternatives to increase performance, such as heterogeneous computer architectures, which use different types of processors. As an illustration, there is current scientific effort towards exascale computing (10^{18} operations/second) by the Exascale Computing Project (ECP) from the US Department of Energy. These exascale systems will rely on heterogeneous architectures in order to be able to achieve their goal. To fully exploit all the advantages of these high-performance architectures, computer programs need to be efficient (highly parallelized) and portable, *i.e.* simulation codes need to be hardware independent so that they can run on novel super-computing clusters without needing to write separate codes for each architecture.

What is the motivation behind this drive for exascale and performance in HPC? Compute-intensive and large-scale simulations in fields like plasma physics, fusion, and astrophysics would benefit greatly from more operations per second and increased efficiency. Reducing the run-time of these simulations could also allow to go to higher resolution and accuracy. This can in turn lead to new scientific observations, such as plasma behaviour and turbulence in the ITER tokamak fusion reactor, which requires expensive high-resolution simulations with $\sim 10^{10}$ computational particles [7].

In this general context, our goal is to design a performance portable Particle-in-Cell (PIC) software. PIC simulations are mainly used in fields such as plasma physics, astrophysics, and accelerator physics in order to simulate the self-consistent dynamics of charged particles. The Independent Parallel Particle Layer (IPPL) is a PIC library which serves as the backend of the particle accelerator simulation tool Object-oriented Parallel Accelerator Library (OPAL) [1]. The goal is to update this already existing PIC library, IPPL 1.0, by interfacing it with a hardware abstraction library, Kokkos [2], which allows code to be written in an architecture independent way so that it can easily be ported. Furthermore, this updated version, IPPL 2.0, makes use of Highly Efficient FFT for Exascale (HeFFTe), a library which targets exascale computing platforms and provides highly efficient and scalable Fast Fourier Transforms (FFTs) [3].

Currently, our focus is on electrostatic PIC. To show the performance and applicability of IPPL 2.0, a suite of mini-apps is developed, known as A pLasma Physics mINiapp for Exascale (ALPINE). The mini-apps include simulations of the self-consistent dynamics of charged particles inside Penning Trap and of the Hose Instability, both well-studied topics in plasma physics.

While there are many aspects to the PIC code, the aim of this particular project is to develop a solver for the Poisson equation with open boundary conditions to be integrated into IPPL 2.0. For open boundary conditions, the integral approach to the problem is better suited, and our solver is based on FFTs for the rapid evaluation of the otherwise computationally expensive volume integral. Various algorithms are explored, such as the quintessential Hockney-Eastwood method [8], or the more novel Vico-Greengard method [9]. Thanks to Kokkos, both of these methods are implemented in the context of the solver and have been tested on both Central Processing Units (CPUs) and Graphics Processing Units (GPUs). The solver also makes use of the Message Passing Interface (MPI) in order to implement inter-nodal communication between processors with no shared memory. This allows to run on more processors and therefore have more computing power and memory, allowing to reach larger problem sizes. The solver is benchmarked on a test problem (a Gaussian charge density profile). Finally, we seek to demonstrate the applicability and performance of the solver through an implementation of the Hose Instability.

Chapter 2

Numerical Methods

In this section, the numerical methods which are used in the project are presented. First, we give a brief overview of the Particle-in-Cell technique for the simulation of kinetic plasmas. Then, we talk about two methods which allow to numerically solve Poisson's equation with open boundary conditions: Hockney-Eastwood, which has been the standard algorithm for many years, and Vico-Greengard, a newer approach to the problem which has been implemented in a performance portable way for the first time. Both are FFT-based methods. Finally, the metric used to verify the correctness and accuracy of the implemented algorithms is presented.

2.1 Particle-in-Cell

The Particle-in-Cell method consists in modelling a cloud of plasma particles by a macro-particle, as otherwise there would be too many particles to account for in the plasmas of interest if we want to model physical phenomena. These macro-particles are tracked in phase space as they evolve over time. The time evolution is determined by Newton's equations of motion, in which the electromagnetic fields come into play for the force field. In order to compute these fields, the charge density of the particles is computed on the grid points of a fixed mesh (**scatter**), on which we solve for the electromagnetic fields (**field solve**). These are then interpolated to the macro-particle locations (**gather**) in order to compute the force which will drive the particles to their new position at the next time-step (**particle push**) [10]. Figure 2.1 shows the PIC setup. The PIC loop consists of the scatter, solve, gather, and push, and is repeated at each time-step until the end of the simulation, schematically shown in Figure 2.2. The reason one can substitute particles by macro-particles is because the Lorentz force, which governs charged particle dynamics, depends on the charge-to-mass ratio of the particles.

Let $\vec{r}$ be the position of a macro-particle with mass m and charge q, $\vec{p} = m\gamma\vec{v}$ its relativistic momentum, $\vec{v}$ its velocity, and $\gamma = \frac{1}{\sqrt{1-|\vec{v}|^2/c^2}}$ the Lorentz factor, where c is the speed of light.

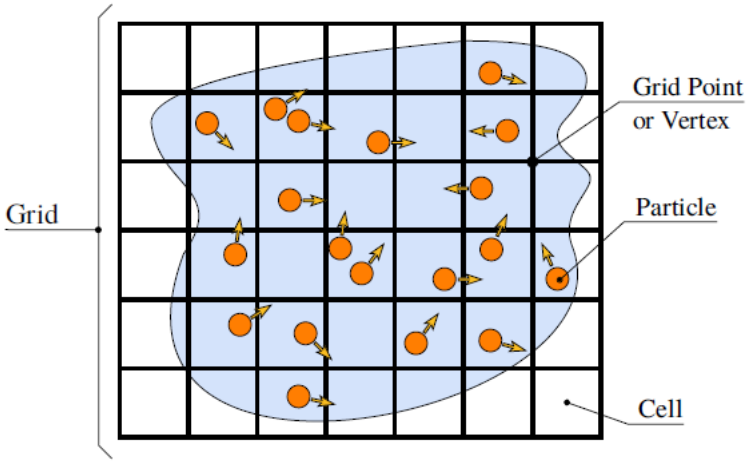

Figure 2.1: Particle-in-Cell model, with particles and mesh [10].

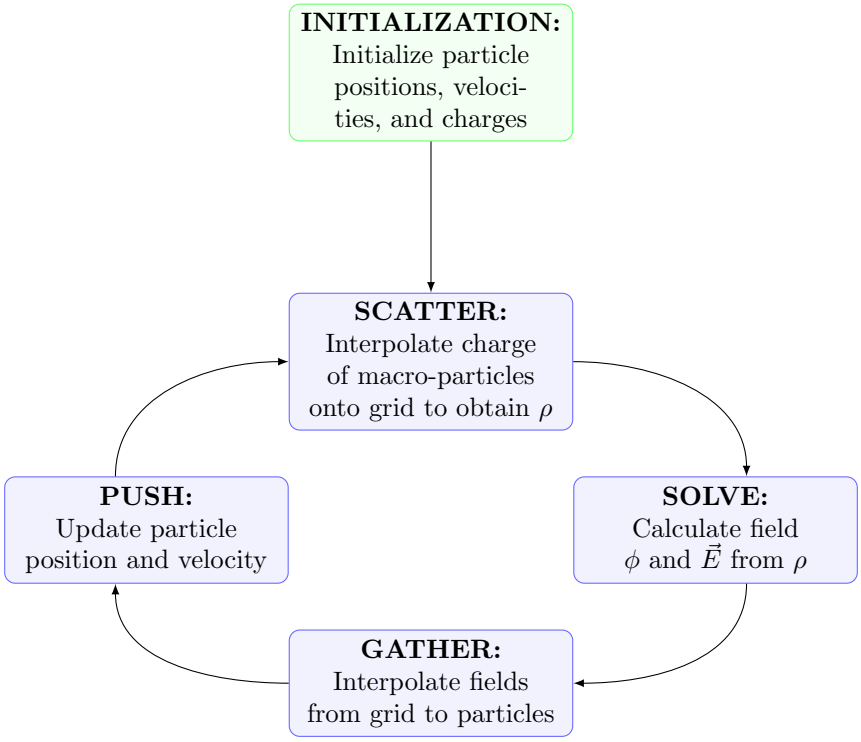

Figure 2.2: The PIC loop, which is repeated every time-step after the initialization.

The equations of motion are given by the Newton-Lorentz equations [1]:

$$\frac{d\vec{r}}{dt} = \frac{\vec{p}}{m\gamma}$$
$$\frac{d\vec{p}}{dt} = q(\vec{E} + \frac{\vec{p}}{m\gamma} \times \vec{B}).$$

The above are integrated using the leapfrog algorithm [11] in the particle push phase, to evolve all of the macro-particles. The fields $\vec{E}$ and $\vec{B}$ are the sum of the self-fields of the particles and any external fields that may be applied, $\vec{E}_{ext}$ and $\vec{B}_{ext}$. In the electrostatic case, the magnetic self-field is zero in the non-relativistic regime. In the case of relativistic particles, a magnetic self-field appears due to the moving charges.

Let $\vec{x} \in \mathbb{R}$ represent a position for fields (not the same as the macro-particle positions). At each time-step, the $\vec{E}(\vec{x})$ field can be computed using the charge density $\rho(\vec{x})$ by means of the Poisson equation, which relates $\rho(\vec{x})$ to the scalar potential $\phi(\vec{x})$:

$$\Delta\phi(\vec{x}) = -\frac{\rho(\vec{x})}{\epsilon_0}$$
$$\vec{E}(\vec{x}) = -\vec{\nabla}\phi(\vec{x}),$$

where ϵ_0 is the dielectric constant. From the $\vec{E}(\vec{x})$ field on the mesh one can obtain the E-field $\vec{E}$ at each macro-particle position $\vec{r}$ through interpolation (in the gather phase).

The focus of this project is to implement a field solver for the Poisson equation on the mesh for open boundary conditions ($\phi \to 0$ as $|\vec{x}| \to \infty$). As already mentioned, these are most naturally dealt with using integral-based methods. In this case, the potential can be written as a convolution [12]:

$$\phi(\vec{x}) = \int G(\vec{x} - \vec{x}')\rho(\vec{x}')d\vec{x}', \tag{2.1}$$

where $G(\vec{x}) = -\frac{1}{4\pi|\vec{x}|}$ is the solution of $\Delta\phi = -\delta(\vec{x})$ with $\phi \to 0$ as $|\vec{x}| \to \infty$, known as Green's function for the Poisson problem with open boundary conditions.

Let the mesh on which the fields are defined be of size $[0, L_x] \times [0, L_y] \times [0, L_z]$, and the number of grid points in the computational grid be $N_x \times N_y \times N_z$, such that the mesh spacing is $h_x = L_x/N_x$, $h_y = L_y/N_y$, $h_z = L_z/N_z$ in the corresponding directions. Then, $\vec{x} = (ih_x, jh_y, kh_z)$ represents a position on the mesh, where $i = \frac{1}{2}, ..., N_x - \frac{1}{2}$, $j = \frac{1}{2}, ..., N_y - \frac{1}{2}$, $z = \frac{1}{2}, ..., N_z - \frac{1}{2}$, since we use a cell-centered scheme for the fields. The convolution 2.1 can be discretized on the field mesh:

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

If this is solved using brute force, the computational cost of the operation is $\mathcal{O}((N_x N_y N_z)^2)$ [13]. The cost of the operation can be brought down by using Fourier transforms, thanks to the convolution theorem [14]:

Theorem 1 *Let $f(\vec{x})$ and $g(\vec{x})$ be two functions. We denote the Fourier transform with $\mathcal{F}$, and the inverse Fourier transform as $\mathcal{F}^{-1}$. The Fourier transforms of f and g are defined as:*

$$F(\vec{s}) = \mathcal{F}(f)(\vec{s}) = \int_{\mathbb{R}^3} f(\vec{x}) e^{-i2\pi\vec{s}\cdot\vec{x}} d\vec{x},$$
$$G(\vec{s}) = \mathcal{F}(g)(\vec{s}) = \int_{\mathbb{R}^3} g(\vec{x}) e^{-i2\pi\vec{s}\cdot\vec{x}} d\vec{x},$$

where $\vec{s} \in \mathbb{R}^3$ is known as the wave-vector.

We denote the convolution of f and g as $f * g$:

$$(f * g)(\vec{x}) = \int f(\vec{x}') g(\vec{x} - \vec{x}') d\vec{x}'$$

The convolution theorem states that a convolution is a multiplication in Fourier space:

$$\mathcal{F}(f * g) = \mathcal{F}(f) \cdot \mathcal{F}(g) \implies f * g = \mathcal{F}^{-1}(\mathcal{F}(f) \cdot \mathcal{F}(g)).$$

One can apply the above theorem to the convolution $\rho * G$, such that the dominating cost in the operation is that of the Fourier transform (the rest is just a point-wise multiplication). If we use fast algorithms for the Fourier transforms, *e.g.* the Fast Fourier Transform (FFT) [15], the cost is drastically reduced to $\mathcal{O}(N\log N)$, where $N = N_x N_y N_z$. To incorporate FFTs in solving the Poisson equation for open boundary conditions, the most commonly used algorithm is that of Hockney-Eastwood [16], explained in the next section.

2.2 Hockney-Eastwood Method

The Hockney-Eastwood method consists in replacing the convolution by a cyclic convolution, such that we can make use of 3-dimensional FFTs to make the computation more efficient. Let the physical domain be $[0, L_x] \times [0, L_y] \times [0, L_z]$, and the mesh the same as defined above. The steps of the algorithm are the following [8]:

1. Double the computational grid in the non-periodic directions (those which have open boundary conditions). If all boundary conditions are open, this means we will have a grid of size $2N_x \times 2N_y \times 2N_z$.
2. The source term $\rho(\vec{x})$, which is defined on $\vec{x} \in [0, L_x) \times [0, L_y) \times [0, L_z)$, is then zero-padded to be defined on the full doubled domain. So for $\vec{x} \in [0, 2L_x) \times [0, 2L_y) \times [0, 2L_z)$, we define $\rho_2(\vec{x})$ such that:
$$\rho_2(\vec{x}) = \begin{cases} \rho(\vec{x}), & \text{if } \vec{x} \in [0, L_x) \times [0, L_y) \times [0, L_z) \\ 0, & \text{otherwise.} \end{cases}$$
3. The Green's function $G(\vec{x})$ also needs to be extended on the doubled domain. In order to

do so, we circular shift it and make it periodic, such that the new function $G_2(\vec{x})$ satisfies:

$$G_2(\vec{x}) = \begin{cases} G(\vec{x}), & \text{if } \vec{x} \in [0, L_x) \times [0, L_y) \times [0, L_z) \\ G(2L_x - x, y, z), & \text{if } \vec{x} \in [L_x, 2L_x) \times [0, L_y) \times [0, L_z) \\ G(x, 2L_y - y, z), & \text{if } \vec{x} \in [0, L_x) \times [L_y, 2L_y) \times [0, L_z) \\ G(x, y, 2L_z - z), & \text{if } \vec{x} \in [0, L_x) \times [0, L_y) \times [L_z, 2L_z) \\ G(2L_x - x, 2L_y - y, z), & \text{if } \vec{x} \in [L_x, 2L_x) \times [L_y, 2L_y) \times [0, L_z) \\ G(2L_x - x, y, 2L_z - z), & \text{if } \vec{x} \in [L_x, 2L_x) \times [0, L_y) \times [L_z, 2L_z) \\ G(x, 2L_y - y, 2L_z - z), & \text{if } \vec{x} \in [0, L_x) \times [L_y, 2L_y) \times [L_z, 2L_z) \\ G(2L_x - x, 2L_y - y, 2L_z - z), & \text{otherwise.} \end{cases}$$

The Green's function is singular at the origin, so we need to regularize it. In this work, we choose to do so by replacing the singularity by $G(\vec{0}) = -1/4\pi$.

4. Now, assume ρ_2 and G_2 periodic with period $2(L_x, L_y, L_z)$. Now we can Fourier transform ρ_2 and G_2 (after discretization) using FFTs, such that the convolution becomes a simple multiplication. In this case, the potential on the doubled domain is given by ϕ_2:

$$\phi_2 = h_x h_y h_z \text{FFT}^{-1}\{\text{FFT}\{\rho_2\} \times \text{FFT}\{G_2\}\}.$$

5. The physical solution for the potential is obtained by restricting the double-grid solution on the physical domain, *i.e.* $\phi(\vec{x}) = \phi_2(\vec{x})$ for $\vec{x} \in [0, L_x) \times [0, L_y) \times [0, L_z)$. The proof of the equivalence in the physical domain between the cyclic convolution and the original convolution can be found in [12, p. 12].

The computational cost of the operation needed to solve the Poisson equation has been reduced to $\mathcal{O}((8N)\log(8N))$, where $N = N_x N_y N_z$. The trade-off is the increased memory requirement to store the fields ρ_2 and G_2 on the doubled domain. The Hockney-Eastwood method is second-order accurate. Thus, if high accuracy is desired, the computation becomes expensive quickly. It could be advantageous to explore other methods with spectral accuracy, such as the new Vico-Greengard method [17].

2.3 Vico-Greengard Method

Vico, Greengard, and Ferrando describe a new method which only differs from the above by the choice of Green's function, and offers spectral accuracy for smooth functions, *i.e.* convergence faster than any fixed power of the mesh spacing h [9].

For the case of the Poisson equation, in which the differential operator is the Laplace operator, the choice of Green's function is given to be the following:

$$G_L(\vec{s}) = 2\left(\frac{\sin(L|\vec{s}|/2)}{|\vec{s}|}\right)^2,$$

where $\vec{s}$ is Fourier wavevector, and G_L is defined in Fourier space. L is the size of the truncation window, which should be chosen to be bigger than the maximum distance between any two points in the computational domain [9]. If the domain is $[0, L_x] \times [0, L_y] \times [0, L_z]$, then we should choose $L > \sqrt{L_x^2 + L_y^2 + L_z^2}$ *i.e.* one should set $L = \alpha \cdot \sqrt{L_x^2 + L_y^2 + L_z^2}$, with $\alpha > 1$. For this study, we set $\alpha = 1.1$ (truncation window 10% larger than the maximum distance between any two points in the domain).

Now, we have the Green's function on the Fourier domain. To not lose any information when doing the inverse Fourier transform of this Green's function, it should be done on a $4N_x \times 4N_y \times 4N_z$ grid, due to the oscillatory nature of G_L. Indeed, since we are carrying out a circular convolution and we want it to give the same result as the linear convolution, we need a doubled grid (as explained by Hockney and Eastwood [16, p. 213]), so at minimum a doubling of grid points is required.

The other factor 2 is explained by the sinusoidal in G_L. To avoid errors when sampling a periodic signal, one should use a sampling rate f_s twice larger than the maximum frequency f_{max} of that signal *i.e.* $f_s > 2f_{max}$ to not lose any information [18]. This statement applies when computing the (inverse) FFT of an oscillating signal, such as G_L, which contains a sinusoidal. One needs a two-times finer mesh order to have a sampling rate twice larger than the frequency content.

So to avoid information loss, the inverse should be taken on the fourfold grid. If all the convolution computation were to be done on the fourfold grid, *i.e.* the source term would also need to be zero-padded on the fourfold domain, it would become prohibitive in memory, making the method unviable for computational simulations. However, this can be bypassed by computing solely then inverse transform of G_L on the fourfold grid as part of a pre-computation step, then restricting it to the doubled grid of size $2N_x \times 2N_y \times 2N_z$, getting G_2. The restricted Green's function G_2 is then stored to be used every time the Poisson solve is called in the PIC loop, such that the fourfold mesh is never needed again. This is only true if a fixed mesh is maintained in the course of the simulation; if the mesh spacing changes, the pre-computation needs to be done again.

Since we already have the pre-computed Green's function G_2 on the doubled domain, we can follow the same steps as Hockney-Eastwood, skipping the Green's function computation:

1. Double the computational grid in the non-periodic directions (those which have open boundary conditions). If all boundary conditions are open, this means we will have a grid of size $2N_x \times 2N_y \times 2N_z$.

2. The source term $\rho(\vec{x})$, which is defined on $\vec{x} \in [0, L_x) \times [0, L_y) \times [0, L_z)$, is then zero-padded to be defined on the full doubled domain. So for $\vec{x} \in [0, 2L_x) \times [0, 2L_y) \times [0, 2L_z)$, we define ρ_2 such that:
$$\rho_2(\vec{x}) = \begin{cases} \rho(\vec{x}), & \text{if } \vec{x} \in [0, L_x) \times [0, L_y) \times [0, L_z) \\ 0, & \text{otherwise.} \end{cases}$$

3. We already have the pre-computed Green's function on the doubled grid G_2.

4. Assume ρ_2 and G_2 periodic with period $2(L_x, L_y, L_z)$, so that we can Fourier transform ρ_2 and G_2 (after discretization) using FFTs such that the convolution becomes a simple

multiplication. In this case, the potential on the doubled domain is given by ϕ_2:

$$\phi_2 = h_x h_y h_z \text{FFT}^{-1}\{\text{FFT}\{\rho_2\} \times \text{FFT}\{G_2\}\}.$$

5. The physical solution for the potential is obtained by restricting the double-grid solution on the physical domain, *i.e.* $\phi(\vec{x}) = \phi_2(\vec{x})$ for $\vec{x} \in [0, L_x) \times [0, L_y) \times [0, L_z)$.

In this case, the resulting algorithm is similar in complexity to Hockney-Eastwood, the sole difference being that we need $\sim 8\times$ more memory for the pre-computation step in Vico-Greengard. This could be a problem on GPU architectures, which usually have less memory available, but in the case of a sufficiently smooth density distribution, the number of grid points required to obtain accurate results will be much lower with Vico-Greengard than with Hockney-Eastwood. The choice of algorithm is therefore best left to the user depending on the characteristics of the problem and the amount of memory available.

2.4 Measuring correctness: Convergence tests

The above algorithms are implemented in our FFT-based Poisson solver in the context of IPPL 2.0. We verify our implementation by means of simple convergence tests with known analytical functions, using the method of exact solutions.

The method of exact solutions consists in verifying the implementation of a differential equation solver by choosing a function, computing its analytical solution, and seeing whether the solution from the implementation approximates the analytical solution. The error between the analytical solution and the computed one should converge to zero as the mesh spacing decreases, and the order of convergence should correspond to the order of the method implemented in the solver. To obtain the theoretically optimal order of convergence one needs to choose smooth analytic functions [19]. This method, along with the method of manufactured solutions, are important to identify coding mistakes and verify that the correct equations are being solved by the implementation [20]. For the Poisson equation, we choose ρ such that the solution ϕ can be analytically computed.

The first problem we choose is the Gaussian source:

$$\rho = -\frac{1}{\sqrt{(2\pi)^3}} \exp\left(-\frac{r^2}{2\sigma^2}\right),$$

such that the exact solution is given by [17]:

$$\phi_{exact} = \frac{1}{4\pi r} \text{erf}\left(\frac{r}{\sqrt{2}\sigma}\right),$$

where $r = \sqrt{(x-\mu)^2 + (y-\mu)^2 + (z-\mu)^2}$ and $\mu = 0.5$, such that the Gaussian peak is at the center of the computational box (three-dimensional box of side $L = 1$). The standard deviation is chosen to be $\sigma = 0.05$.

For our second test, we choose the source related to the gravitational potential of a sphere, which we use for the purpose of comparison with the results shown in [21], and for demonstrating

that Vico-Greengard fails to converge faster than Hockney-Eastwood when the sources are non-smooth functions. The source term is given by the following expression:

$$\rho = \begin{cases} 4\pi G, & \text{if } r \leq 1 \\ 0, & \text{if } r > 1, \end{cases}$$

and the exact solution is [21]:

$$\phi_{exact} = \begin{cases} -\frac{2}{3}\pi G \cdot (3 - r^2), & \text{if } r \leq 1 \\ -\frac{4}{3}\pi G \cdot \frac{1}{r}, & \text{if } r > 1, \end{cases}$$

where $G = 6.67408 \cdot 10^{-11}$ $\mathrm{m^3 kg^{-1} s^{-2}}$ is the gravitational constant, r is as above and we choose $\mu = 1.2$ this time to center the distribution in the box (3D box of sides $L = 2.4$).

The metric used to measure the relative error between the exact solution ϕ_{exact} and the computed solution is the $L2-$norm:

$$\text{relative error} = \frac{||\phi - \phi_{exact}||_2}{||\phi_{exact}||_2}.$$

We solve using the two sources above for increasing number of grid points, and see whether we reproduce the correct order of convergence for each method. We also double-check that the algorithms are correctly implemented by comparing with the relative $L2$-error obtained for the same grid size using a MATLAB prototype code (*c.f.* Appendix A). If the difference between both errors is below computer precision (*i.e.* $< 10^{-14}$), we can conclude that it is correctly implemented.

The implementation also includes the computation of the electric field, $\vec{E} = -\vec{\nabla}\phi$. The user can choose how the E-field is calculated: applying a finite difference gradient to ϕ, or through Fourier differentiation, which is when you apply the gradient in Fourier space such that it becomes a multiplication with the wave-vector: $\vec{\nabla}\phi = \mathcal{F}^{-1}(\vec{s} \cdot \mathcal{F}(\phi))$.

If the first method is chosen, the E-field accuracy will depend on the accuracy of the finite difference gradient (second-order in the case of central finite difference). For Fourier differentiation, $\vec{E}$ will retain the same accuracy as ϕ. For Vico-Greengard, it would be advantageous to choose this method since then the E-field computation will also be spectrally accurate [17].

2.5 Application: Hose Instability

The Hose Instability is a phenomenon in which a beam of electrons propagating though a plasma channel starts to oscillate in the transverse plane, with the oscillations growing with time, due to the interaction between the plasma channel and the electrons [22]. Both the plasma channel and the electron beam suffer from hosing. It is particularly of interest in the plasma wake-field accelerator community, as it can arise in plasma wake-field experiments and impact results and performance [23].

We seek to model the Hose Instability using [24] as a reference. For the Hose Instability to appear, we need to take into account transverse electric and magnetic fields. However, currently

only electrostatic PIC is implemented in IPPL 2.0. To circumvent this, we decide to constrain ourselves to the case of a coasting electron beam with constant longitudinal velocity v_z. This way, by Lorentz boosting to the rest frame of the beam, we can solve the Poisson equation to find the E-field just as we would in the electrostatic case, and then compute the resulting B-field due to the inverse Lorentz boost back to the laboratory frame. If the speed of a charged particle is $v = |\vec{v}|$, then the laboratory frame E-field and B-field ($\vec{E}$ and $\vec{B}$) are related to the beam rest frame ones ($\vec{E}'$ and $\vec{B}'$) by [25]:

$$\begin{aligned} E_x &= \gamma E'_x \\ E_y &= \gamma E'_y \\ E_z &= E'_z \\ B_x &= -\frac{\beta}{c} E_y \\ B_y &= \frac{\beta}{c} E_x \\ B_z &= 0, \end{aligned}$$

where $\beta = \frac{v}{c}$ and $\gamma = \frac{1}{\sqrt{1-\beta^2}}$ are relativistic factors, and c is the speed of light.

In the transverse ($\perp$) direction, the Lorentz force equation is:

$$\frac{d\vec{p}_\perp}{dt} = q\left(\vec{E}_\perp + \frac{v_z}{c}\vec{e}_z \times \vec{B}_\perp + \frac{\vec{v}_\perp}{c} \times B_z \vec{e}_z\right).$$

For the Hose Instability simulation, we will use the parameters presented in [24], with all open boundary conditions so that the solver implemented in this project can be used:

- The electron beam is initialized as a Gaussian distribution of radius $R_{rms} = 5$ cm in the transverse plane, with a small sinusoidal perturbation of amplitude $10^{-2} R_{rms}$. The pulse length of the electron beam in the z-direction is 2 μs, and the longitudinal motion is relativistic, with $\gamma = 25$.
- The ion channel is initialized to be coaxial with the electron beam, Gaussian distributed in the transverse plane with same radius as the beam, and uniformly distributed over the entire box length in the longitudinal direction. The ion species is H_2O^+.
- We add an external B-field in the longitudinal direction, of magnitude 830 G.

After setting these initial conditions and letting the simulation evolve, one should be able to observe oscillation of the electron beam in the transverse plane due to the appearance of the Hose Instability. The measurements should be done at the end of a 50 m drift length.

Chapter 3

Computer Architecture

"It's hardware that makes a machine fast. It's software that makes a fast machine slow" [26]. The choice of software is therefore important when aiming for performance, and especially if we are targeting exascale computing. It is equally important to specify the hardware used for performance studies, as performance depends on the computer architecture. In this section, we present the specifications of the hardware used, as well as a brief description of the libraries that have been incorporated into IPPL 2.0.

3.1 Hardware used

3.1.1 CPUs

The simulations are run at Paul Scherrer Institut's Merlin6 computing cluster. Each node (*a.k.a.* processor) consists of an Intel Xeon Gold 6152 processor with 2 sockets, 44 cores and 2 threads per core. It is possible to run with only one thread per core or with hyperthreading (using both threads). The memory available is 384 GB, of which 352 GB are usable. Infiniband technology allows for inter-node connectivity, with 2400 Gbps bandwidth among nodes in the same chassis and 1200 Gbps between different chassis.

3.1.2 GPUs

In terms of GPUs, the latest NVIDIA DGX A100 system is used. This system contains 8 NVIDIA A100 Tensor Core GPUs, with a total memory of 320 GB (40 GB per GPU). There is a Dual AMD Rome 7742 CPU with 128 cores and 1 TB system memory. Communication between GPUs is provided by NVLink, which has a direct bandwidth of 600 Gbps.

3.2 Software Tools

3.2.1 Kokkos

The Kokkos project [2], developed mainly at Sandia National Laboratories, aims at abstracting the hardware part of parallel programming. As part of the Exascale Computing Project, the ultimate goal is to prepare scientific computing programs for new exascale infrastructures such as Aurora [27]. This library allows us to incorporate portability into IPPL, such that Kokkos takes care of making it run on specific computing architectures, like CPUs or GPUs, primarily within a single node. Figure 3.1 shows the structure of the Kokkos ecosystem, with Kokkos Core being the programming model at the heart of the project. Kokkos Kernels provides linear algebra software, while Kokkos Tools provides profiling and debugging tools. Kokkos Remote Spaces provide some support for inter-node parallelism.

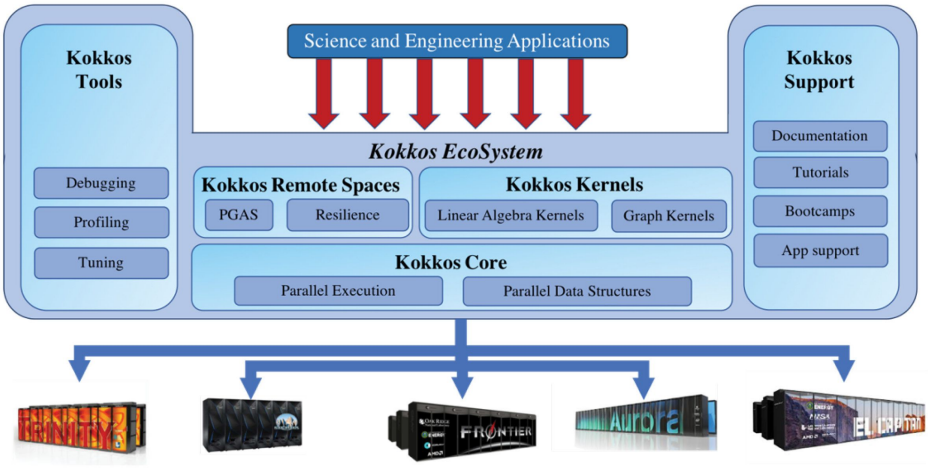

Figure 3.1: The Kokkos ecosystem [28].

The Kokkos Core programming model allows for the user to write code as instructions using common parallel programming patterns, decide the execution spaces of the instructions, and where the data is stored. On top of that, Kokkos optimizes data access for the specific architecture, such that performance is not compromised. In IPPL 2.0, we use Kokkos multi-dimensional arrays, known as Views, for the fields and particles, as Kokkos can then optimize access to field and particle data according to the architecture. In terms of parallel execution patterns, the most commonly present in the Poisson solver implementation and corresponding test scripts are parallel for loops and reductions.

A parallel for loop is a for loop in which iterations are divided between processes. To do so, it requires each iteration to be independent of the others. In the case where one wants to compute an operation such as an addition or multiplication, a reduction should be used, as the iterations are not independent of each other. For example, in the Poisson solver convergence test scripts, parallel for loops (`Kokkos::parallel_for`) are used for initialization of fields (Kokkos Views), while reductions (`Kokkos::parallel_reduce`) are used for the computation of the relative error

norm. Listings 3.1 and 3.2 show examples of these, respectively. The code snippets are taken from `TestGaussian_hockney.cpp`, which is the convergence test for a Gaussian charge distribution using Hockney-Eastwood (*c.f.* Appendix A for access to the script). Some liberties were taken when presenting the code snippets here for the sake of brevity. The `MDRangePolicy` in these code snippets indicates the execution policy; in these cases, we want a 3D nested loop to be able to access field data in the Views, and we provide the iteration range.

```
typename field::view_type& view_rho = rho.getView(); // View for rho field

// Ghost/guard cells are cells added to the local sub-domain in a process
// such that field data is available locally for finite differences methods
const int nghost = rho.getNghost(); // number of ghost cells

// The local sub-domain contained in the process running the code
// (as the domain is divided between processes)
const auto& ldom = layout.getLocalNDIndex(); // sub-domain in this process

Kokkos::parallel_for("Assign rho field",
    Kokkos::MDRangePolicy<Kokkos::Rank<3>>({nghost, nghost, nghost},
                                           {view_rho.extent(0) - nghost,
                                            view_rho.extent(1) - nghost,
                                            view_rho.extent(2) - nghost}),
            KOKKOS_LAMBDA(const int i, const int j, const int k){
                // go from the local indices of this process to global indices
                const int ig = i + ldom[0].first() - nghost;
                const int jg = j + ldom[1].first() - nghost;
                const int kg = k + ldom[2].first() - nghost;

                // compute the physical position (cell-centered mesh)
                double x = (ig + 0.5) * hr[0] + origin[0];
                double y = (jg + 0.5) * hr[1] + origin[1];
                double z = (kg + 0.5) * hr[2] + origin[2];

                // assign the gaussian at that position to the rho View
                view_rho(i, j, k) = gaussian(x, y, z);
});
```

Listing 3.1: Example of parallel for loop for initialization of ρ with a Gaussian distribution.

```
fieldE = fieldE - exactE; // difference between computed and exact E-field
auto view_fieldE = fieldE.getView(); // View for E-field (vector)
unsigned int d = 0; // component of E-field (0 = x component)
double temp = 0.0; // variable which will hold the value of the reduction

Kokkos::parallel_reduce("Vector errorNr reduce",
    Kokkos::MDRangePolicy<Kokkos::Rank<3>>({nghost, nghost, nghost},
                                           {view_fieldE.extent(0) - nghost,
                                            view_fieldE.extent(1) - nghost,
                                            view_fieldE.extent(2) - nghost}),
            KOKKOS_LAMBDA(const int i, const int j, const int k, double& valL) {
                // square the difference between exact and computed Efield
                // for L2 norm relative error computation
                double myVal = pow(view_fieldE(i, j, k)[d], 2);

                // sum over the squares
                // this will give numerator of relative error
                valL += myVal;

            }, Kokkos::Sum<double>(temp)); // temp = sum(valL of each process)
```

Listing 3.2: Example of reduction needed as part of the relative error norm computation for E_x. The variable `nghost` is the same as in Listing 3.1.

This higher level of programming avoids the need to rewrite code every time we want to port our code to a different architecture. Without Kokkos, the estimated amount of code one would need to rewrite when porting is $\sim 10\%$ [29]. So if IPPL 2.0 ($\approx$ 20k lines of code) did not use Kokkos, we would need to rewrite ≈ 2000 lines of code every time we need to run on a different architecture machine. Thanks to Kokkos, we avoid this work and can run on many architectures without need for changes to the source code.

3.2.2 HeFFTe

Highly Efficient FFT for Exascale, also known as heFFTe [3], is a library aiming at optimising FFT computation for exascale, and is also part of the Exascale Computing Project. The main attraction of the library is their implementation of the distributed 3D FFT algorithm that addresses the inter-node communication needed in order to perform large-scale FFTs, and shows good performance for these large problem sizes [3].

There are many choices of parameters for the FFT computation left to the user. For example, depending on the architecture, the user can choose the communication scheme which results in better performance: *point-to-point* communication (pairwise interaction between specific processes) or *all-to-all* communication (all processors send or receive data) [30].

One can also choose between *reorder* or *no reorder*. The three-dimensional FFT is obtained by performing FFTs in each dimension, so between these FFTs a data reshape is needed (*c.f.* Figure 3.2). By specifying reorder, one chooses to reorder the data to be contiguous in memory during the reshape step.

Furthermore, there is a choice of two different decompositions of the computational box which are needed to compute the FFT: *slab* and *pencil*. These are shown in Figure 3.2, which also shows the algorithm followed by heFFTe. More information is found in [3].

3.2.3 MPI

IPPL needs a system for the communication between processors in a distributed memory model, as Kokkos can only access data within the processor where it is executing (unless we use Kokkos Remote Spaces). For this, the Message Passing Interface (MPI) library is used. MPI allows the user to move data between processors with control on the communication scheme used [30]. The heFFTe library presented above also makes use of MPI for inter-processor communication. In MPI, we refer to each process as a *rank*. In our experiments, we use one MPI rank per node.

For both the Hockney-Eastwood and Vico-Greengard algorithms, communication between nodes is needed for the zero-padding of the source term ρ to obtain ρ_2.

If we run the program on n ranks, the computational domain is divided into n boxes, each of which is stored in a separate rank. Now, if we define a new computational domain of double the size in each dimension, this computational domain is also divided into n boxes in order to be distributed among the n ranks. Both the physical and doubled domains are divided into n sub-domains. However, the sub-domains are not equivalent, as one is a physically larger domain that the other. Hence, each rank contains different sub-domains of the $N_x \times N_y \times N_y$ grid and the $2N_x \times 2N_y \times 2N_z$. When writing ρ to ρ_2, we need communication to get the sub-domains

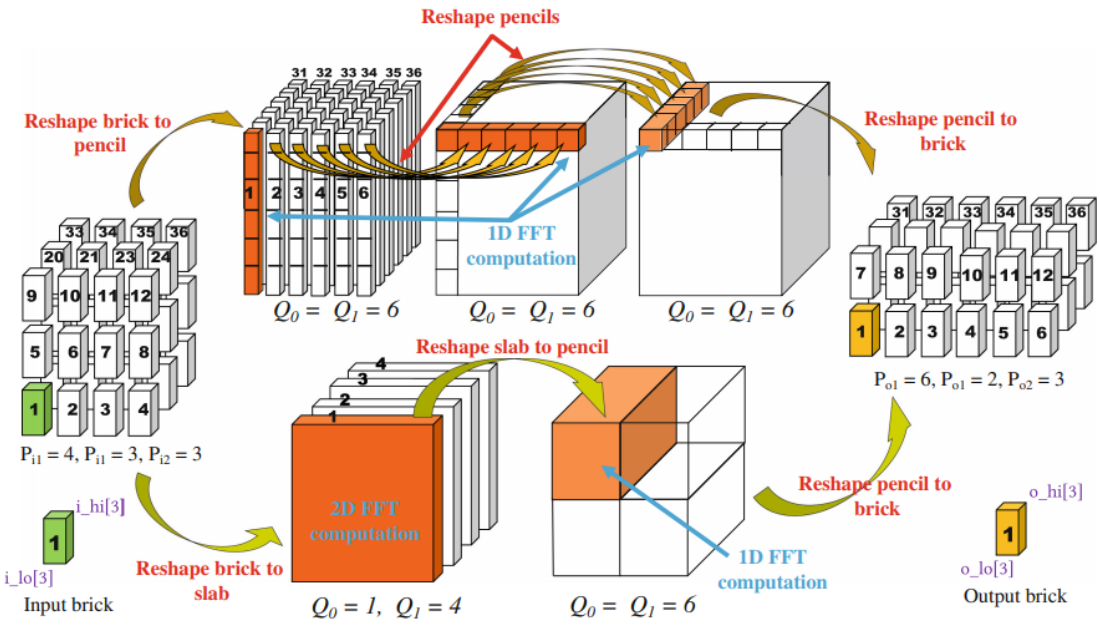

Figure 3.2: The heFFTe algorithm for pencil and slab decompositions [3].

of the field ρ to the rank which corresponds to the same sub-domain in ρ_2. A diagram of the problem is shown in Figure 3.3.

The implementation of this assignment of a field from the physical grid to a doubled grid is non-trivial. The approach followed is shown in Algorithm 1. All ranks will encounter this algorithm and execute it, such that the end result will be what is desired (Figure 3.3). A similar algorithm is used for the restriction from the doubled grid to the physical grid ($\phi_2 \rightarrow \phi$).

3.3 Measuring performance: Scaling studies

Performance can be studied using scaling tests, which show how the timings of a program vary as you vary number of ranks and/or amount of work. There are two types of scaling: strong and weak.

3.3.1 Strong Scaling

When conducting strong scaling studies, one keeps the problem size fixed and varies the number of ranks used. Ideally, it is expected that doubling the resources makes the time to solution decrease by half [31].

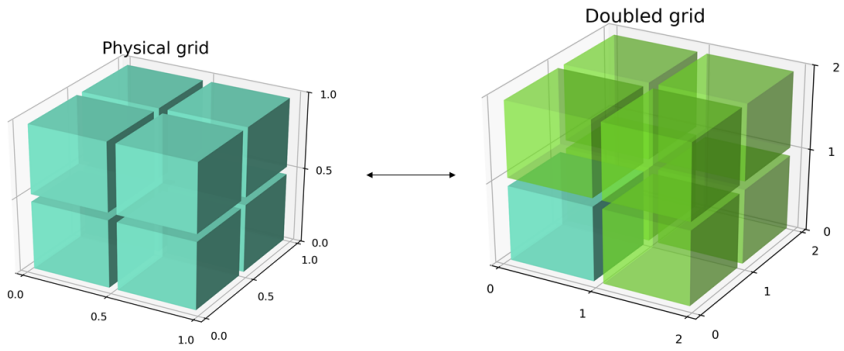

Figure 3.3: Communication is needed to go from the physical grid ($N_x \times N_y \times N_y$) to the doubled grid ($2N_x \times 2N_y \times 2N_z$). On the right, the blue cube indicates the physical sub-domain of the doubled grid, whereas the green cubes are those where ρ will need to be zero-padded.

In the case of strong scaling, you can quantify it using speedup S_p and efficiency E_p^s [32]:

$$S_p = \frac{T_1}{T_p} \qquad E_p^s = \frac{S_p}{p},$$

where p is the number of processes, T_1 is the most optimal time to solution on one process, and T_p is the time to solution on p processes.

3.3.2 Weak Scaling

For weak scaling studies, the problem size per rank is kept constant such that when you increase the number of ranks, the problem size also increases. Ideally, the time to solution should stay constant, which would mean that the work increases proportionally to the number of processes [31]. Nevertheless, such ideal performance is very hard to achieve, especially when there is MPI communication involved, since communication time will increase as the number of ranks increases.

In this case, we can only define efficiency as a way to quantify performance (notation as above) [32]:

$$E_p^w = \frac{T_1}{T_p}.$$

Algorithm 1 Assignment from physical to doubled grid. The bounds of the sub-domains contained in each rank are globally known.

$n \leftarrow$ total number of ranks
$myRank \leftarrow$ ID of this rank
$domains1 \leftarrow$ bounds of sub-domains of physical grid
$domains2 \leftarrow$ bounds of sub-domains of doubled grid

for $i \leftarrow 1, n$ **do**
  if $domains2[i]$ touches $domains1[myRank]$ **then**
    intersection $\leftarrow$ intersect $domains2[i]$ with $domains1[myRank]$
    Pack $domains1[myRank][intersection]$.
    Send packed data to rank i.
  end if
end for

for $i \leftarrow 1, n$ **do**
  if $domains1[i]$ touches $domains2[myRank]$ **then**
    $intersection \leftarrow$ intersect $domains1[i]$ with $domains2[myRank]$
    Receive data from rank i.
    Unpack data into $domains2[myRank][intersection]$.
  end if
end for

Chapter 4

Results and Discussion

4.1 Convergence Tests

The implementation of Hockney-Eastwood and Vico-Greengard needs to be verified, so convergence tests are performed with a smooth Gaussian as the source, for both the potential and the electric field. Figure 4.1 shows the convergence tests for the potential for both algorithm choices and on both CPU and GPU. The difference with the MATLAB prototype code is below computer precision (10^{-14}) for all data points. Figure 4.2 shows a similar plot for the electric field computation. The grid sizes used were $4^3, 8^3, 16^3, 32^3, 64^3$ and 128^3.

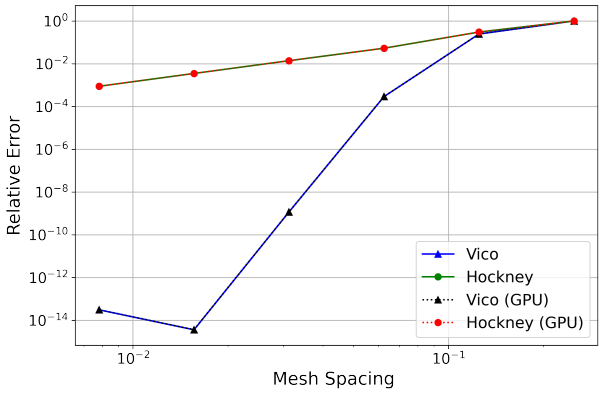

Figure 4.1: Convergence test, Hockney and Vico, for the potential.

As can be seen, Hockney-Eastwood converges with second order and Vico-Greengard spectrally, consistent with what is expected. The CPU and GPU results match, assuring us of a proper portable implementation. There are slight fluctuations in the relative error once it dips below 10^{-14}, since it is computer precision.

The second test case is that of the sphere, which is non-smooth. We ran the Hockney-Eastwood solver with the following problem sizes: $48^3, 144^3, 288^3, 384^3$ and 576^3. These have

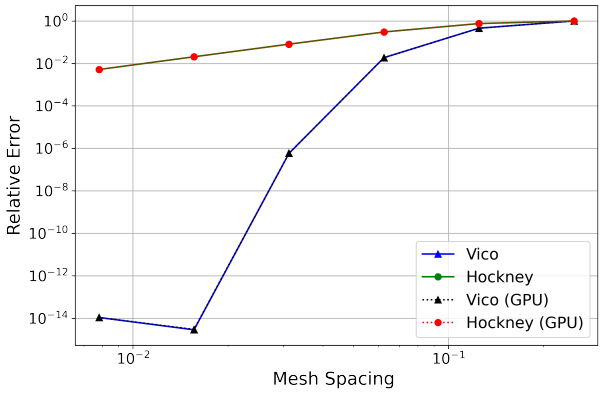

Figure 4.2: Convergence test, Hockney and Vico, for the electric field.

been chosen in order to compare the results of the implemented Hockney-Eastwood Poisson solver with what is shown in [21], where they also test the Hockney-Eastwood algorithm on this problem. The relative errors for the potential are shown in Figure 4.3. For the purpose of comparison with [21, fig. 7], the $L1$-norm relative error is shown in this figure. The data-points are in agreement with [21, fig. 7], which is a verification that the implemented solver is correct and has the expected decrease in accuracy due to the non-smoothness of ρ.

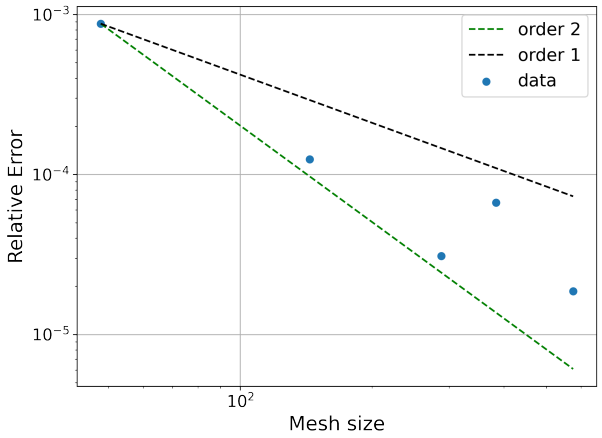

Figure 4.3: Convergence test for potential of a sphere with the Hockney-Eastwood method. For this non-smooth problem, Hockney-Eastwood is between first-order and second-order accuracy.

Figure 4.4 shows the relative error between analytical and computed potential using both Hockney-Eastwood and Vico-Greengard. As can be seen, Vico-Greengard does not converge spectrally, but rather follows the pattern of Hockney-Eastwood, demonstrating the drawback of Vico-Greengard for non-smooth distributions. The user should therefore choose appropriately between the two depending on the properties of the problem at hand. Table 4.1 shows a comparison between the time and memory costs of both methods for a specific accuracy, in which it is evident that for a sufficiently smooth problem, Vico-Greengard ends up being more economic. This should be taken with a grain of salt for arbitrary problems though, as shown by the case of the spherical potential.

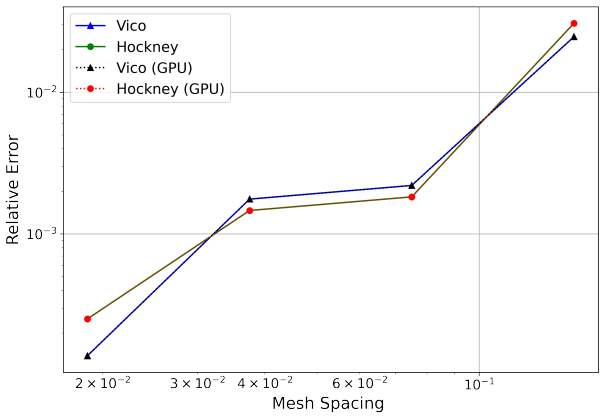

Figure 4.4: Convergence test for potential of a sphere, for both Hockney-Eastwood and Vico-Greengard, for grid sizes from 16^3 to 128^3.

	Hockney	Vico	Hockney	Vico
Relative Error	Gridsize		Memory [MB]	
$\sim 10^{-1}$	8^3	8^3	$\sim 10^{-1}$	~ 1
$\sim 10^{-2}$	32^3	-	$\sim 10^{1}$	-
$\sim 10^{-3}$	64^3	-	$\sim 10^{2}$	-
$\sim 10^{-4}$	128^3	16^3	$\sim 10^{3}$	$\sim 10^{1}$
$\sim 10^{-9}$	$\sim (2^{15})^3$	32^3	$\sim 10^{10}$	$\sim 10^{2}$

Table 4.1: For a specific accuracy, comparison between both methods of the grid sizes, memory, and time taken for one solve, for a smooth source.

4.2 Scaling studies: Hockney-Eastwood

Scaling studies are conducted to measure the performance of the solver. We run 5 iterations of the solver and time it, for varying number of ranks. As mentioned before, the parameters for the FFT computation via heFFTe are to be user-provided, so we run the scaling tests with all the possible combinations of FFT parameters to judge which parameters are most suited to our problem and computer architecture.

First of all, we ran a strong scaling study for the Hockney-Eastwood algorithm for a 256^3 problem size. The number of nodes increases in powers of 2, up to 16 CPUs, and up to 8 GPUs. Appendices B & C show the results for all FFT parameter combinations on CPUs and GPUs respectively. Figure 4.5, which compares the total time of all FFT parameter combinations on CPU, demonstrates that all parameter combinations scale well, and the best one is using a pencil decomposition, all-to-all communication, and reordering. However, the differences between parameter choices is not drastic. On the other hand, on GPUs, we observe sub-optimal performance (Figure 4.6). Moreover, there is a set of parameters which takes noticeably less time than others. The combination with best speedup seems to be slab decomposition, all-to-all communication,

with reordering. However the best choice for lower time to solution is slab, point-to-point, no reordering.

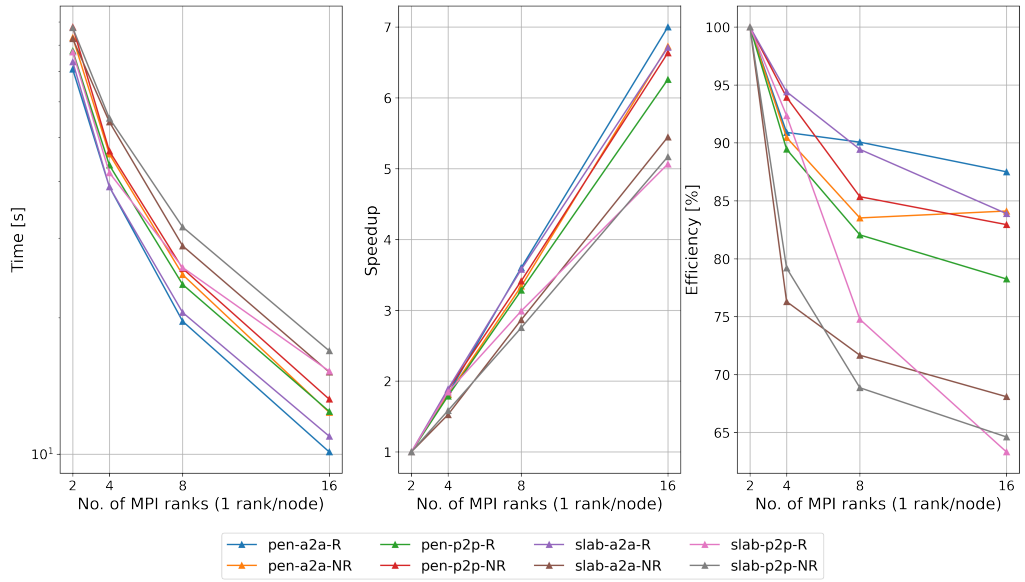

Figure 4.5: Time taken on CPUs for 5 iterations of the Poisson solver using the Hockney-Eastwood algorithm for different combinations of heFFTe parameters and increasing number of nodes. The problem size is fixed to $N_x N_y N_z = 256^3$. Speedup and Efficiency are also plotted.

While performance seems to approach ideal performance on CPU, on GPU poor scaling is observed. The reason for this was identified to be a lack of sufficient work for the GPU to show scaling. Therefore, a new scaling study is performed for a 512^3 problem size. The memory required for this problem size exceeds the memory available on 2 GPUs, therefore in this case the GPU scaling tests start from 4 GPUs. There are also two parameter combinations which exceed available memory, so they are not considered (slab, all-to-all, with reordering and slab, point-to-point, with reordering). Appendix D shows the results, and Figure 4.7 shows a comparison of scaling results between heFFTe parameter combinations. From this figure, we see that the pencil decomposition, point-to-point communication, with reordering shows the best performance, and has a low time to solution. The hypothesis that lack of sufficient work caused poor performance on GPU is also confirmed, as now we observe scaling and up to $\approx 1.6\times$ speedup from 4 to 8 GPUs.

A more detailed breakdown of the pencil, point-to-point with reordering scaling test for $N_x N_y N_z = 512^3$ is shown in Figure 4.8. The most time-consuming tasks are the FFT computations, especially for the E-field, as this consists of three FFTs, one for each dimension. The FFTs scale well, as shown by the speedup graph. The communication kernels increase in time as we increase the number of GPUs, consistent with what is expected since more communication is required if there are more GPUs to communicate between. The time required for the assignment to and from the doubled grid is low compared to the FFT computation time, showing that this communication cost is not prohibitive. The solver initialization time (which includes Green's function computation) is not accounted for in the total time, as it is only a one time cost. However, if we were to change the field mesh size in the course of a simulation, the Green's function

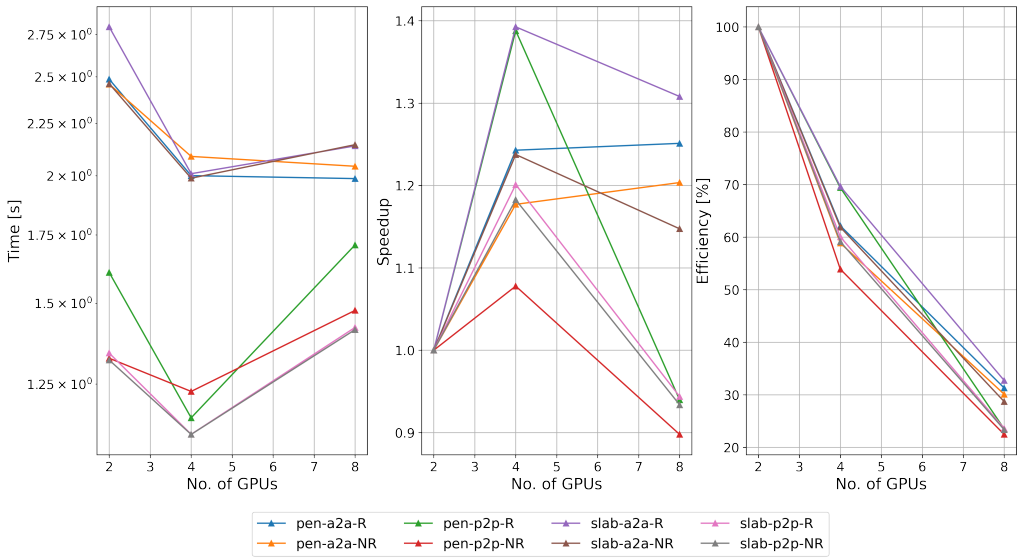

Figure 4.6: Time taken on GPUs for 5 iterations of the Poisson solver using the Hockney-Eastwood algorithm for different combinations of heFFTe parameters and increasing number of nodes. The problem size is fixed to $N_x N_y N_z = 256^3$. Speedup and Efficiency are also plotted.

would need to be recomputed, hence incurring an additional cost, which would be dominated by the FFT computation of the Green's function.

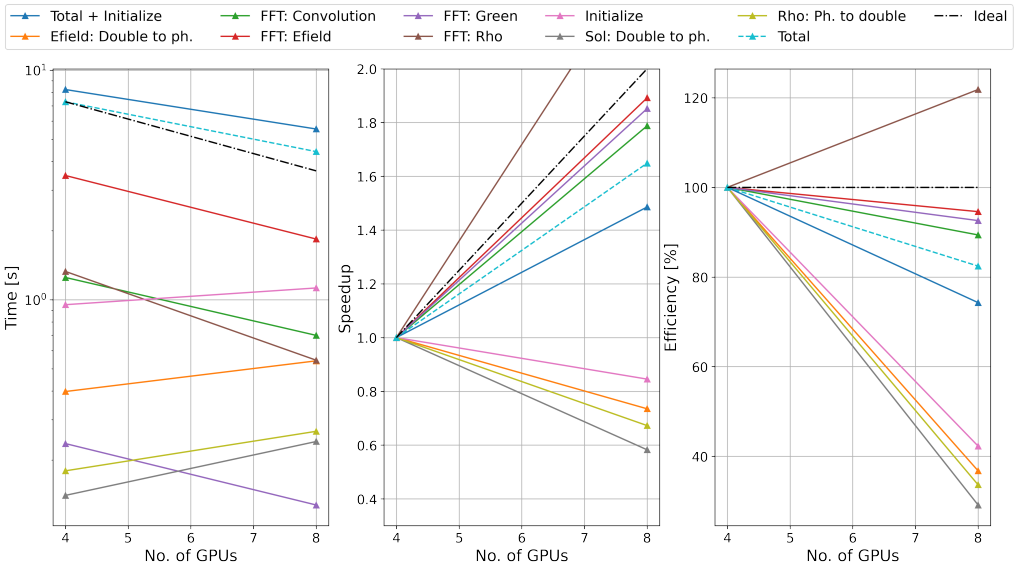

Figure 4.8: Time taken on GPUs for 5 iterations of the Poisson solver using the Hockney-Eastwood algorithm on an increasing number of nodes, with pencil decomposition, point-to-point, with reordering. The problem size is fixed to $N_x N_y N_z = 512^3$. Speedup and Efficiency are also plotted.

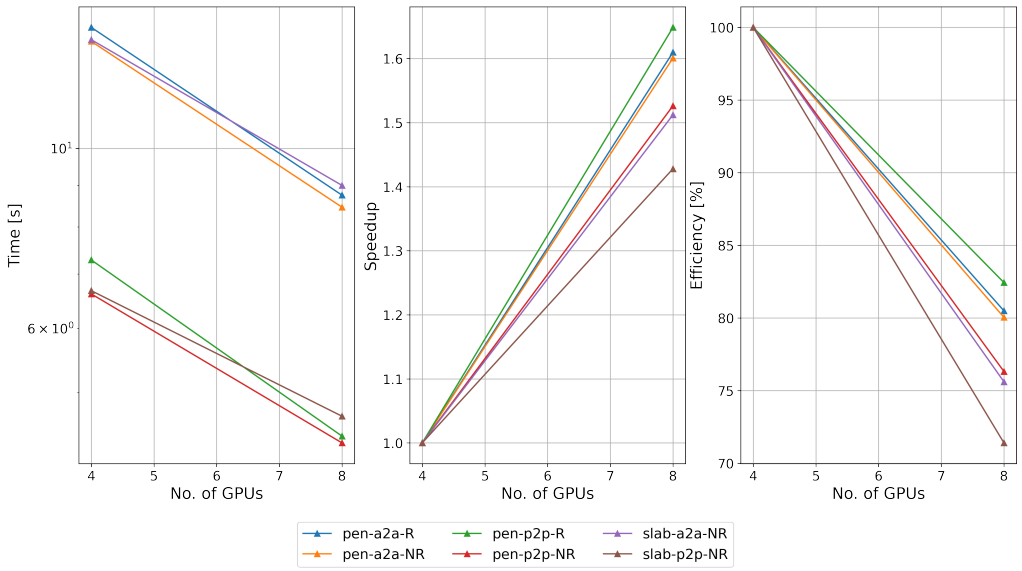

Figure 4.7: Time taken on GPUs for 5 iterations of the Poisson solver using the Hockney-Eastwood algorithm for different combinations of heFFTe parameters and increasing number of nodes. The problem size is fixed to $N_x N_y N_z = 512^3$. Speedup and Efficiency are also plotted.

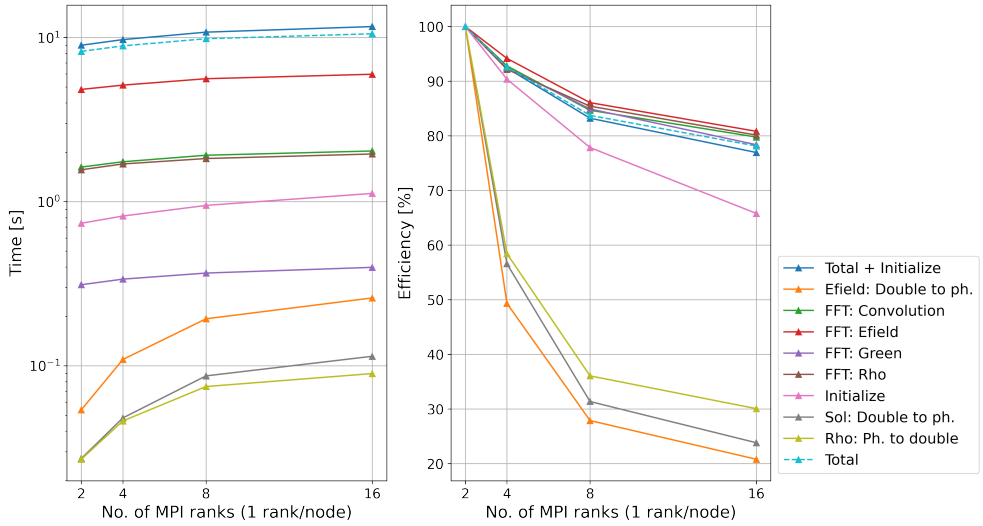

Figure 4.9: Weak scaling study on CPU for Hockney-Eastwood, showing time taken by all kernels for 5 iterations of a solve, as well as the corresponding efficiency. Work per node is kept constant as number of nodes increase, starting from $N_x N_y N_z = 128^3$.

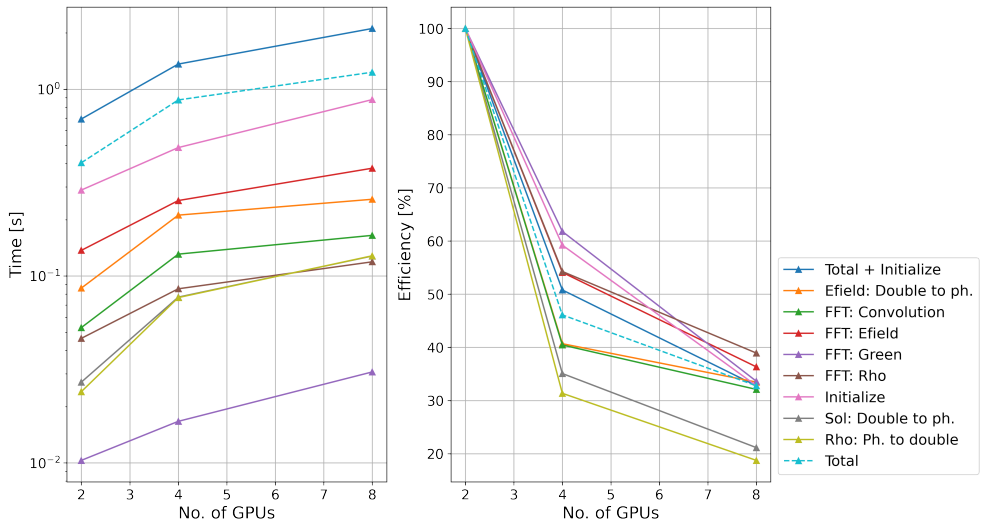

Figure 4.10: Weak scaling study on GPU for Hockney-Eastwood, showing time taken by all kernels for 5 iterations of a solve, as well as the corresponding efficiency.

Weak scaling studies of Hockney-Eastwood are conducted using the optimal parameter choice for CPU and GPU: pencil, all-to-all, with reordering and pencil, point-to-point, with reordering, respectively. On CPU (Figure 4.9) the time to solution stays roughly constant, as does the time for FFT computation. Total efficiency reaches $\approx 80\%$ at 16 nodes. The communication time required for assignment to the doubled grid increases, as expected. On the other hand, no weak scaling is observed on GPU (Figure 4.10), as the total efficiency goes down to about 30% at 8 GPUs. The FFT kernels show similar behaviour, possibly because of lack of sufficient work for GPUs to perform well. Further studies should include running on other computing clusters with more GPUs to investigate this behaviour.

4.3 Scaling studies: Vico-Greengard

The heFFTe parameter choice for Vico-Greengard is done according to the observations from the scaling studies of the Hockney-Eastwood implementation, taking into consideration that Vico-Greengard requires an FFT computation on a mesh $2\times$ larger in each dimension than the Hockney-Eastwood mesh. We choose a problem size of $N_x N_y N_z = 128^3$ (larger problem size will not fit on 2 or 4 GPUs), which corresponds approximately in memory and work to Hockney-Eastwood with 256^3. We will therefore expect deterioration in performance from 4 to 8 nodes. Using the conclusion drawn from Figure 4.6, we decide to choose slab decomposition, all-to-all communication, with reordering for GPU. For CPU, we choose the same configuration with the decomposition changed to pencils. We run weak scaling studies for Vico-Greengard with the same parameters.

Figure 4.11 shows that strong scaling on CPU is very close to ideal. Contrastingly, on GPU (Figure 4.12) the time to solution increases with increasing number of nodes, indicating that scaling is not achieved. This is to be expected as the amount of work is too little for the GPUs,

however we cannot increase the problem size as this would exceed available memory. Similarly to what was seen with Hockney-Eastwood, the FFT computation of the E-field dominates the cost, and on CPU the communication costs required for assignment to and from the doubled grid are well below the FFT costs. Nevertheless, on GPU we see that communication between GPUs for this assignment becomes very expensive, and similar in order of magnitude to the FFT costs. This could be a problem in the future, and we need to further investigate by running larger problem sizes and/or go farther than 8 GPUs on other larger computing clusters.

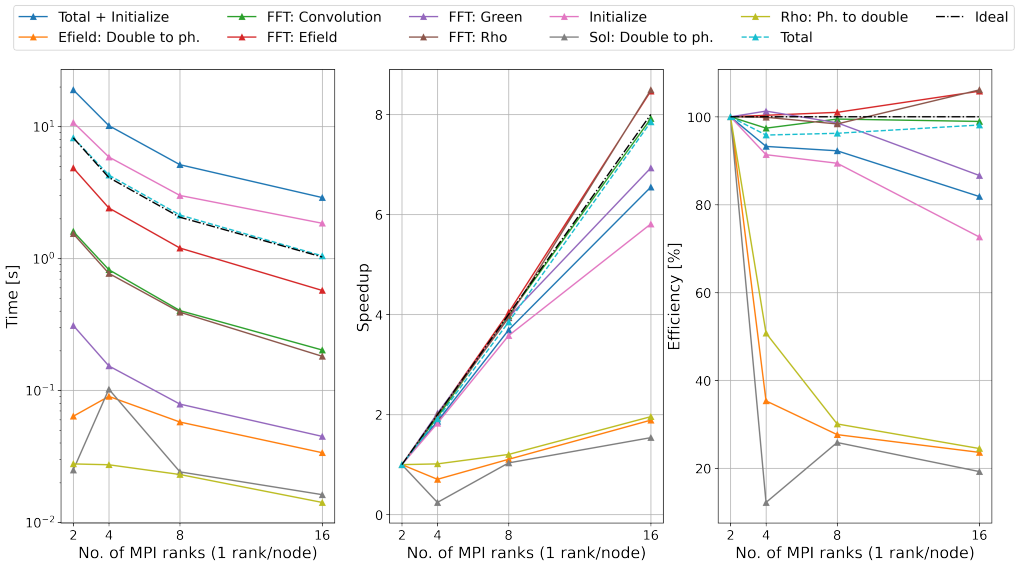

Figure 4.11: Strong scaling study on CPU for Vico-Greengard, showing time taken by all kernels for 5 iterations of a solve, with $N_x N_y N_z = 128^3$. Speedup and Efficiency are also plotted.

As for weak scaling, we see that on CPU the time stays roughly constant (Figure 4.13), and the total efficiency is only reduced to ≈80% for 8 times more nodes, indicating weak scaling is achieved. The FFTs kernels also stay roughly constant, whereas the communication for writing to and from the doubled grid increases in time as the number of nodes increase. This behaviour is expected, as these are not purely parallel kernels and the amount of communication increases with more nodes despite workload staying constant. On GPU, Figure 4.14 shows that weak scaling is not achieved. The communication costs for assignment to and from the doubled grid are also relatively high, as already seen in the strong scaling study on GPU. The same conclusion is reached as with the weak scaling studies for Hockney-Eastwood: further studies on larger GPU clusters are required.

4.4 Hose Instability

An implementation of the simulation for the Hose Instability was produced (*c.f.* Appendix A for access). Unfortunately, the simulation failed to produce any significant results. The self-fields are not correctly computed, causing crashes in the simulation as well as energy non-conservation. The issue is currently being investigated.

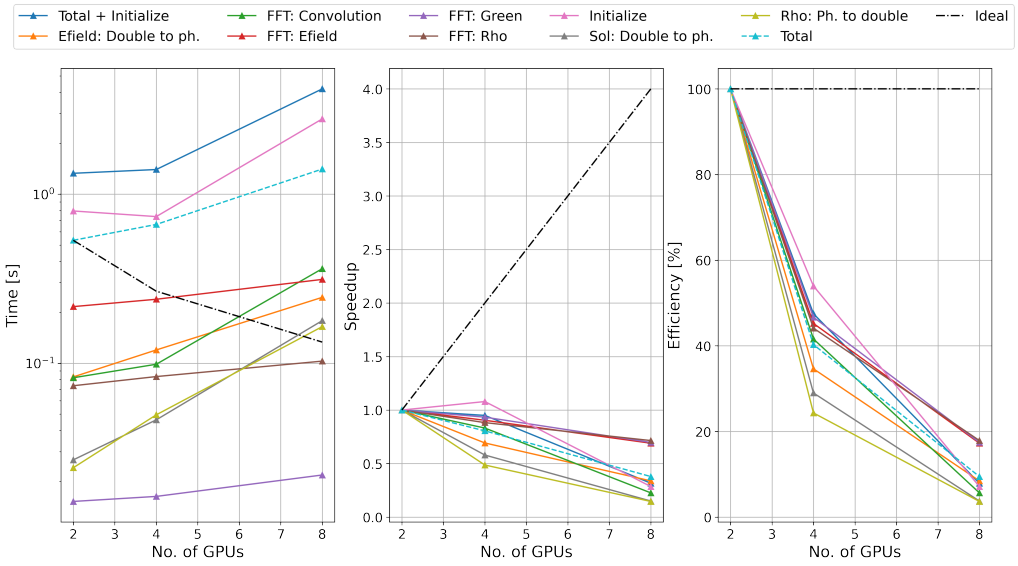

Figure 4.12: Strong scaling study on GPU for Vico-Greengard, showing time taken by all kernels for 5 iterations of a solve, with $N_x N_y N_z = 128^3$. Speedup and Efficiency are also plotted.

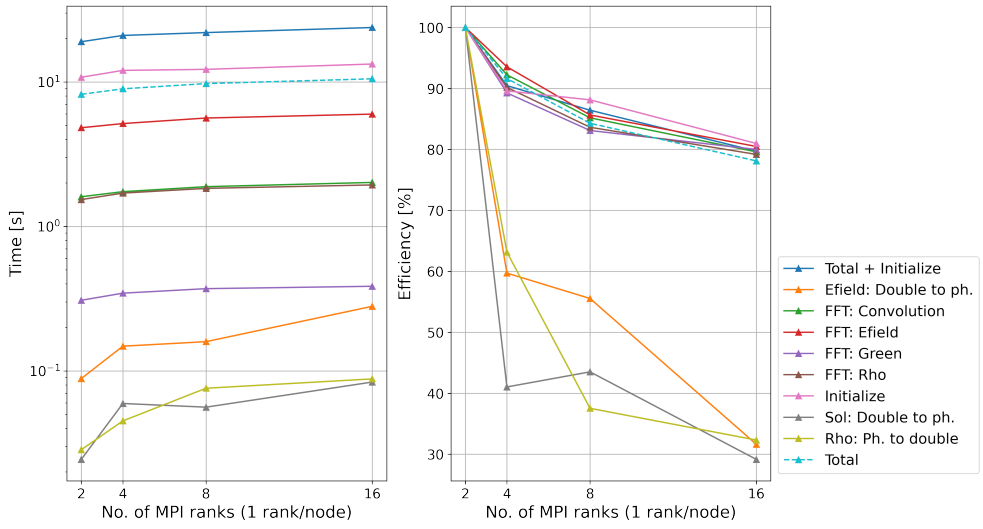

Figure 4.13: Weak scaling study on CPU for Vico-Greengard, showing time taken by all kernels for 5 iterations of a solve, as well as the corresponding efficiency. Problem size is increased proportionally to nodes, starting from $N_x N_y N_z = 128^3$.

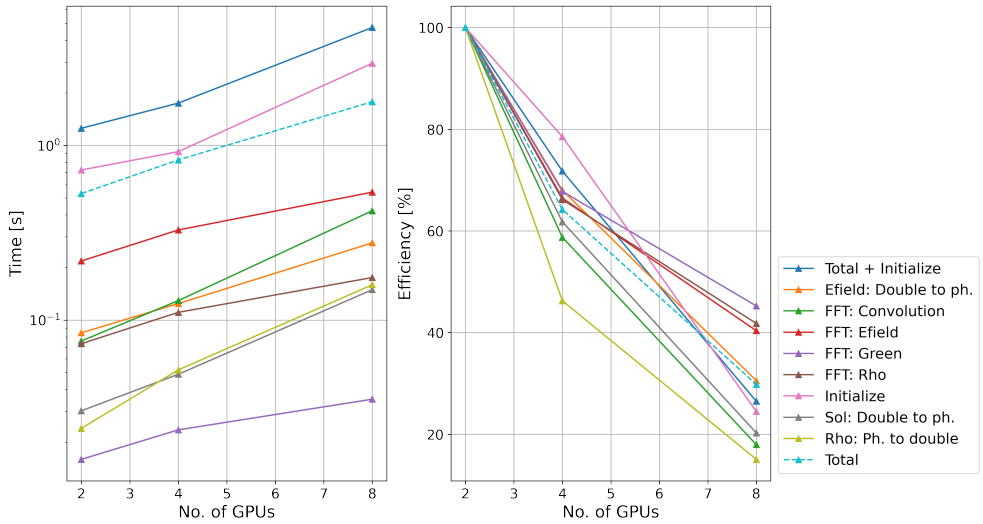

Figure 4.14: Weak scaling study on GPU for Vico-Greengard, showing time taken by all kernels for 5 iterations of a solve, as well as the corresponding efficiency.

Chapter 5

Conclusion and Future Outlook

Overall, the implemented solver showed correctness and, in the case of CPUs, good strong and weak scaling results. There is ample room for improvement in the case of GPU performance. From our presented studies, it is seen that for GPUs to perform well, more workload is needed. Running on larger GPU computing clusters might shed light onto the situation.

We also remark that both implemented methods have their advantages and shortcomings. For sufficiently smooth functions and when high accuracy is desirable, it is seen that choosing Vico-Greengard as the solver method is the best choice. It allows the user to achieve computer precision accuracy without the need for a very fine mesh, albeit at the cost of storage space. However, for non-smooth functions, Vico-Greengard can make no promises of spectral convergence, and seems to converge just as fast as Hockney-Eastwood. In cases such as these, it is better to choose Hockney-Eastwood to avoid unnecessary extra memory costs.

In future work, another algorithm to be explored is James' method [33]. The current bottleneck is memory on GPUs, therefore methods which require less memory should be investigated. James' method allows to circumvent the doubling of the domain, therefore saving on memory costs. To solve the Poisson equation $\Delta\phi = -\rho$, James suggests the following steps [34]:

1. Calculate ψ satisfying $\Delta\psi = -\rho$ with homogeneous Dirichlet boundary conditions. This is achieved via a sine transform in all three directions. The resulting ψ is known as the interior potential.

2. Calculate the image charge distribution q on the boundaries due to the potential ψ using Gauss' law, considering $\psi = 0$ outside the domain. This charge distribution on the surface q is the charge needed to compensate for the original distribution ρ in order to have zero potential at the boundaries.

3. Find the potential θ induced by the screening charge q.

   The potential ψ is the sum of ϕ, the original potential corresponding to ρ, and θ, the potential corresponding to the surface charge q. This is because as explained above, θ (produced by q) and ϕ (produced by ρ) compensate each other to have 0 potential at the boundaries, giving exactly the interior potential ψ.

4. Let us denote with the subscript B the value of a potential at the boundary of the domain. Since $\psi_B = 0$ at the boundaries, we have $\phi_B = -\theta_B$. So by solving $\Delta\phi = -\rho$ with Dirichlet boundary conditions given by $\phi_B = -\theta_B$, one can finally obtain ϕ.

Although it requires more computational cost, we do not need to pay in terms of memory. James' method has already been integrated in other software, such as the magneto-hydrodynamics `Athena++` code [35], where it has shown second-order accuracy and good performance [34]. If this is implemented in IPPL 2.0, it could surpass the attractiveness of Hockney-Eastwood and Vico-Greengard, especially on GPUs, and give the user yet another choice of solver. Finally, the Hose Instability implementation should be completed in future work to show the applicability of the Poisson solver.

Acknowledgements

First and foremost, I cannot express my gratitude enough to my supervisor Andreas Adelmann, for allowing me to be part of the AMAS group and the IPPL 2.0 project. Thank you very much for providing guidance, support, and enthusiasm whenever needed, and always being available for a chat. Thank you for always giving me liberty when deciding the direction of my master thesis, and being a great mentor. I could never have asked for a better supervisor.

I would also like to thank Sriramkrishnan Muralikrishnan, for guiding me in the right direction. Thank you for never getting tired of my incessant questioning and clearing my doubts, I really appreciated your help. Next, I would like to thank Antoine Cerfon, whose aid in understanding the Vico-Greengard algorithm and plasma physics examples was very valuable. Thank you for being available for discussions and sharing your ideas and comments with me. Hopefully we can meet in person one day! I am also grateful to Matthias Frey, for his beneficial input during the weekly meetings, as well as rapidly solving any compilation problems. I would also like to thank Alessandro Vinciguerra, who sent me his shell scripts for the automation of the scaling studies on the cluster, and was a wonderful colleague. Thank you to Andreas Pautz for agreeing to be my supervisor on the EPFL end.

I would like to acknowledge Achim Gsell and Marc Caubet Serrabou, for their help with the git repository of this thesis and Merlin cluster issues, respectively. My sincere thanks also go to the rest of the AMAS group members: Renato, Arnau, Sichen, and Romana, who openly welcomed me into their group and made this past year at PSI unforgettable. I would especially like to thank Renato, my officemate, for answering all my spontaneous questions. Thank you also to Viviana, for helping with administrative issues and being such a positive presence.

Finally, I would like to thank my all friends, who have made my Master years memorable and fun: Arne, Basile, Charlotte, David, Francesca, Manisha, Marko, Max, Mehdi, Myriam, Patrick, Quentin, Soumya, and Valentina. Special thanks to those who proofread my thesis. Last but not least, thank you to my family, for being supportive and always believing in me.

Bibliography

[1] 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," *arXiv:1905.06654 [physics]*, May 2019. arXiv: 1905.06654.

[2] H. Carter Edwards, C. R. Trott, and D. Sunderland, "Kokkos: Enabling manycore performance portability through polymorphic memory access patterns," *Journal of Parallel and Distributed Computing*, vol. 74, July 2014. Institution: Sandia National Lab. (SNL-NM), Albuquerque, NM (United States) Number: SAND-2013-5603J Publisher: Elsevier.

[3] A. Ayala, S. Tomov, A. Haidar, and J. Dongarra, "heFFTe: Highly Efficient FFT for Exascale," in *Computational Science – ICCS 2020* (V. V. Krzhizhanovskaya, G. Závodszky, M. H. Lees, J. J. Dongarra, P. M. A. Sloot, S. Brissos, and J. Teixeira, eds.), Lecture Notes in Computer Science, (Cham), pp. 262–275, Springer International Publishing, 2020.

[4] J. Strickland, "How Moore's Law Works," Feb. 2009. Available at `https://computer.howstuffworks.com/moores-law5.htm`, accessed 25/08/2021.

[5] "Moore's Law: Transistors per microprocessor." Available at `https://ourworldindata.org/grapher/transistors-per-microprocessor`, accessed 11/07/2021.

[6] National Research Council, *The Future of Computing Performance: Game Over or Next Level?* Washington, DC: The National Academies Press, 2011.

[7] F. Hariri, T. M. Tran, A. Jocksch, E. Lanti, J. Progsch, P. Messmer, S. Brunner, C. Gheller, and L. Villard, "A portable platform for accelerated PIC codes and its application to GPUs using OpenACC," *Computer Physics Communications*, vol. 207, pp. 69–82, Oct. 2016.

[8] J. W. Eastwood and D. R. K. Brownrigg, "Remarks on the solution of poisson's equation for isolated systems," *Journal of Computational Physics*, vol. 32, pp. 24–38, July 1979.

[9] F. Vico, L. Greengard, and M. Ferrando, "Fast convolution with free-space Green's functions," *Journal of Computational Physics*, vol. 323, pp. 191–203, Oct. 2016.

[10] X. Saez, A. Soba, J. M. Cela, E. Sanchez, and F. Castejon, "Particle-in-Cell Algorithms for Plasma Simulations on Heterogeneous Architectures," in *2011 19th International Euromicro Conference on Parallel, Distributed and Network-Based Processing*, (Ayia Napa, Cyprus), pp. 385–389, IEEE, Feb. 2011.

[11] J. P. Verboncoeur, "Particle simulation of plasmas: review and advances," *Plasma Physics and Controlled Fusion*, vol. 47, pp. A231–A260, Apr. 2005. Publisher: IOP Publishing.

[12] R. D. Ryne, "On FFT-based convolutions and correlations, with application to solving Poisson's equation in an open rectangular pipe," *arXiv:1111.4971 [physics]*, Nov. 2011.

arXiv: 1111.4971.

[13] J. Qiang, S. Lidia, R. D. Ryne, and C. Limborg-Deprey, "Three-dimensional quasistatic model for high brightness beam dynamics simulation," *Physical Review Accelerators and Beams*, vol. 9, p. 044204, Apr. 2006.

[14] E. W. Weisstein, "Convolution Theorem." Publisher: Wolfram Research, Inc.

[15] J. W. Cooley and J. W. Tukey, "An Algorithm for the Machine Calculation of Complex Fourier Series," *Mathematics of Computation*, vol. 19, no. 90, pp. 297–301, 1965. Publisher: American Mathematical Society.

[16] R. Hockney and J. W. Eastwood, *Computer Simulation Using Particles*. CRC Press, 1988.

[17] J. Zou, E. Kim, and A. J. Cerfon, "FFT-based free space Poisson solvers: why Vico-Greengard-Ferrando should replace Hockney-Eastwood," *arXiv:2103.08531 [physics]*, Mar. 2021. arXiv: 2103.08531.

[18] A. Hero, "Nyquist Sampling Theorem." Available at `https://www.eecs.umich.edu/courses/eecs206/archive/f02/public/lec/lect20.pdf`, accessed 15/07/2021.

[19] M. Jelinek and J. Mahaffy, "The Method of Manufactured Solutions." Available at `https://personal.psu.edu/jhm/ME540/lectures/VandV/MMS_summary.pdf`, accessed on 14/08/2021.

[20] K. Salari and P. Knupp, "Code Verification by the Method of Manufactured Solutions," Tech. Rep. SAND2000-1444, Sandia National Laboratories, June 2000. Available at `https://www.osti.gov/servlets/purl/759450`, accessed on 14/08/2021.

[21] R. D. Budiardja and C. Y. Cardall, "Parallel FFT-based Poisson solver for isolated three-dimensional systems," *Computer Physics Communications*, vol. 182, pp. 2265–2275, Oct. 2011.

[22] C. Huang, W. Lu, M. Zhou, C. E. Clayton, C. Joshi, W. B. Mori, P. Muggli, S. Deng, E. Oz, T. Katsouleas, M. J. Hogan, I. Blumenfeld, F. J. Decker, R. Ischebeck, R. H. Iverson, N. A. Kirby, and D. Walz, "Hosing Instability in the Blow-Out Regime for Plasma-Wakefield Acceleration," *Physical Review Letters*, vol. 99, p. 255001, Dec. 2007.

[23] B. Blue, C. E. Clayton, E. Dodd, K. Marsh, W. B. Mori, C. Joshi, R. Assmann, F.-J. Decker, M. J. Hogan, R. H. Iverson, P. Raimondi, D. Walz, R. H. Siemann, M. Park, P. Muggli, and T. Katsouleas, "Test of the electron hose instability in the E157 experiment," *Proceedings of the 2001 Particle Accelerator Conference, Chicago*, 2001.

[24] T. C. Genoni and T. P. Hughes, "Ion-hose instability in a long-pulse linear induction accelerator," *Physical Review Special Topics - Accelerators and Beams*, vol. 6, p. 030401, Mar. 2003.

[25] D. Tong, "Lectures on Electromagnetism: 4. Electromagnetism and Relativity." Available at `https://www.damtp.cam.ac.uk/user/tong/em.html`, accessed 16/08/2021.

[26] C. Bruce, "Dr. Craig S. Bruce." Available at `http://csbruce.com`, accessed 11/08/2021.

[27] N. Heinonen, "Porting a particle-in-cell code to exascale architectures | Argonne Leadership Computing Facility." Available at `https://alcf.anl.gov/news/porting-particle-cell-code-exascale-architectures`, accessed 16/07/2021.

[28] C. Trott and J. Snyder, "The Kokkos EcoSystem." Available at `https://www.sandia.gov/news/publications/computing_reports/articles/2021/kokkos.html`, accessed on 15/08/2021.

[29] "Kokkos Lecture Series." Available at `https://github.com/kokkos/kokkos-tutorials/tree/main/LectureSeries`, accessed 16/07/2021.

[30] V. Eijkhout, "Parallel Programming in MPI and OpenMP," *Parallel Computing*, p. 373.

[31] V. Eijkhout, R. van de Geijn, and E. Chow, *Introduction To High Performance Scientific Computing*. Zenodo, Apr. 2016.

[32] "Measuring Parallel Scaling Performance - Documentation." Available at `https://www.sharcnet.ca/help/index.php/Measuring_Parallel_Scaling_Performance`, accessed 16/07/2021.

[33] R. A. James, "The solution of poisson's equation for isolated source distributions," *Journal of Computational Physics*, vol. 25, pp. 71–93, Oct. 1977.

[34] S. Moon, W.-T. Kim, and E. C. Ostriker, "A Fast Poisson Solver of Second-Order Accuracy for Isolated Systems in Three-Dimensional Cartesian and Cylindrical Coordinates," *The Astrophysical Journal Supplement Series*, vol. 241, p. 24, Mar. 2019. arXiv: 1902.08369.

[35] J. M. Stone, K. Tomida, C. J. White, and K. G. Felker, "The `Athena++` Adaptive Mesh Refinement Framework: Design and Magnetohydrodynamic Solvers," *The Astrophysical Journal Supplement Series*, vol. 249, p. 4, June 2020.

Appendix A: IPPL library, MATLAB codes, and post-processing

The IPPL library can be found in the following Gitlab repository: `https://gitlab.psi.ch/OPAL/Libraries/ippl`. The wiki of this repository contains information on the structure of IPPL 2.0, as well as installation information for Kokkos and heFFTe.

The Solver fork containing the work for this thesis, including convergence tests and Hose Instability implementation, can be found here: `https://gitlab.psi.ch/mayani_s/ippl-solvers`.

The following repository contains all the post-processing scripts used to obtain the results presented, as well as the MATLAB prototype codes: `https://gitlab.psi.ch/AMAS-members/ippl-sonali`. Intermediate presentations given during the master thesis are also included in this repository.

Instructions on how to install Kokkos and heFFTe, compile IPPL, run the solver test scripts, and use the post-processing scripts to obtain results can be found here: `https://gitlab.psi.ch/AMAS-members/ippl-sonali/-/wikis/home`.

Appendix B: Scaling tests on CPU, 256^3

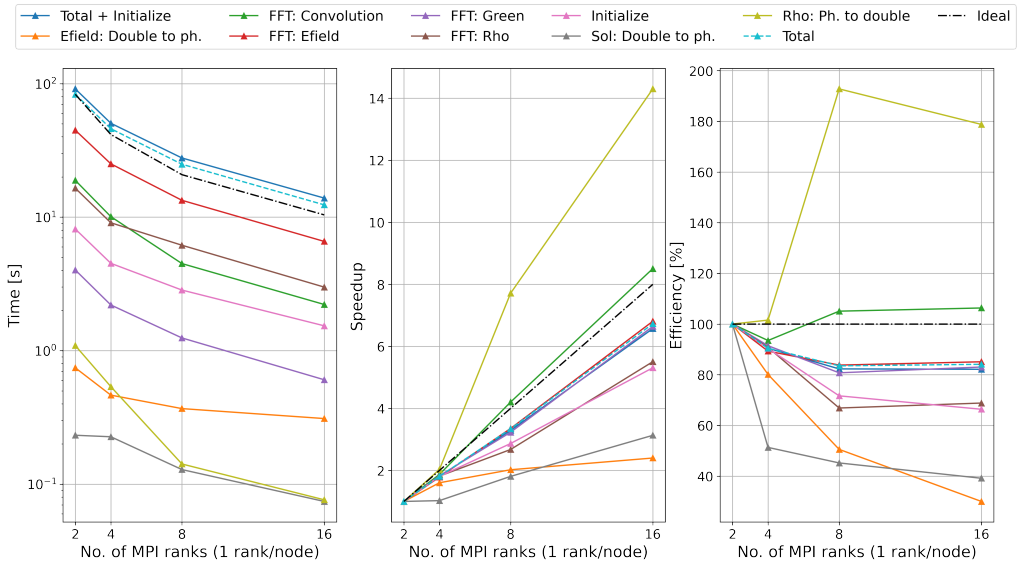

Figure B.1: Strong scaling test for pencil decomposition, all-to-all communication, no reordering.

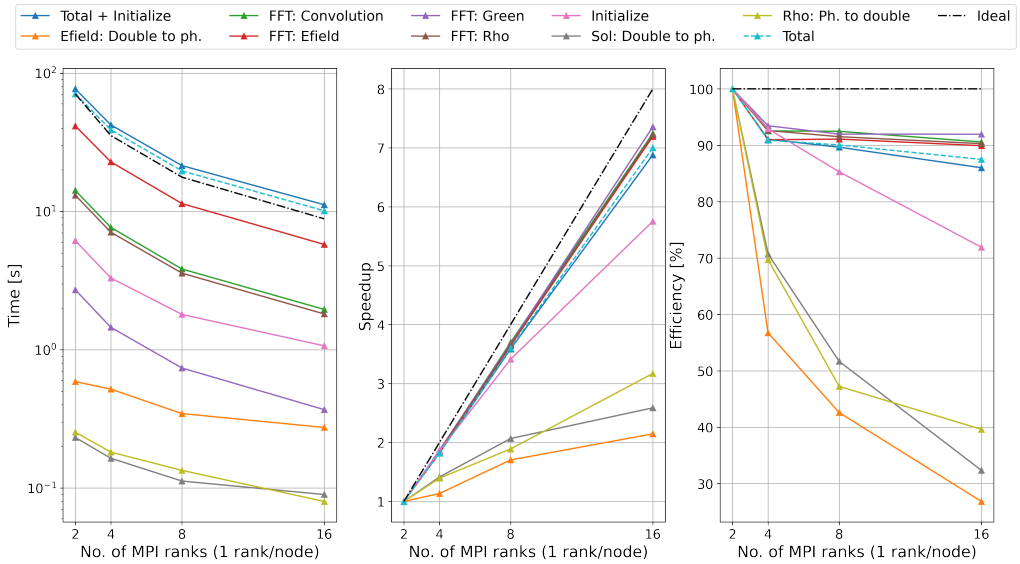

Figure B.2: Strong scaling test for pencil decomposition, all-to-all communication, with reordering.

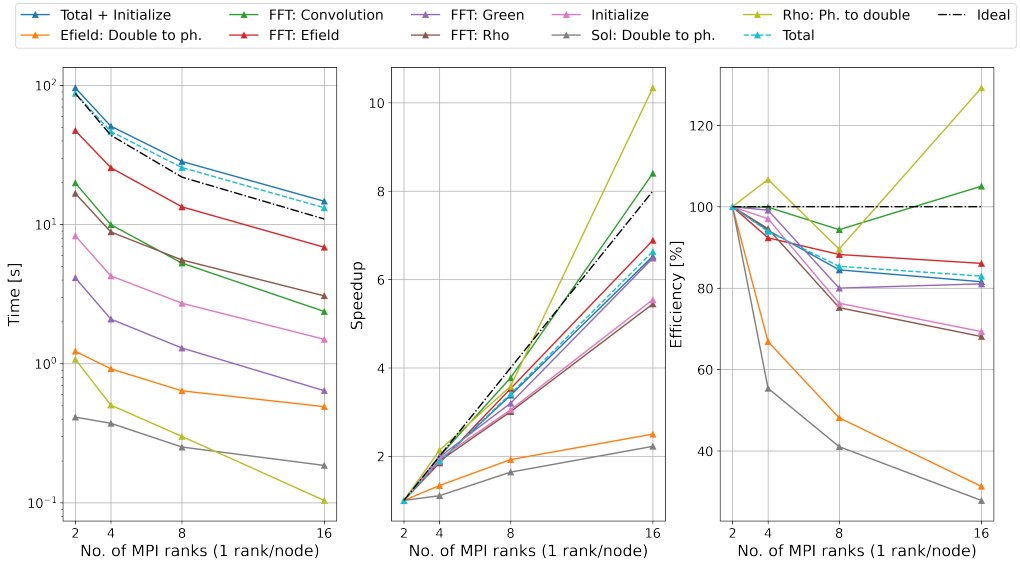

Figure B.3: Strong scaling test for pencil decomposition, point-to-point communication, no reordering.

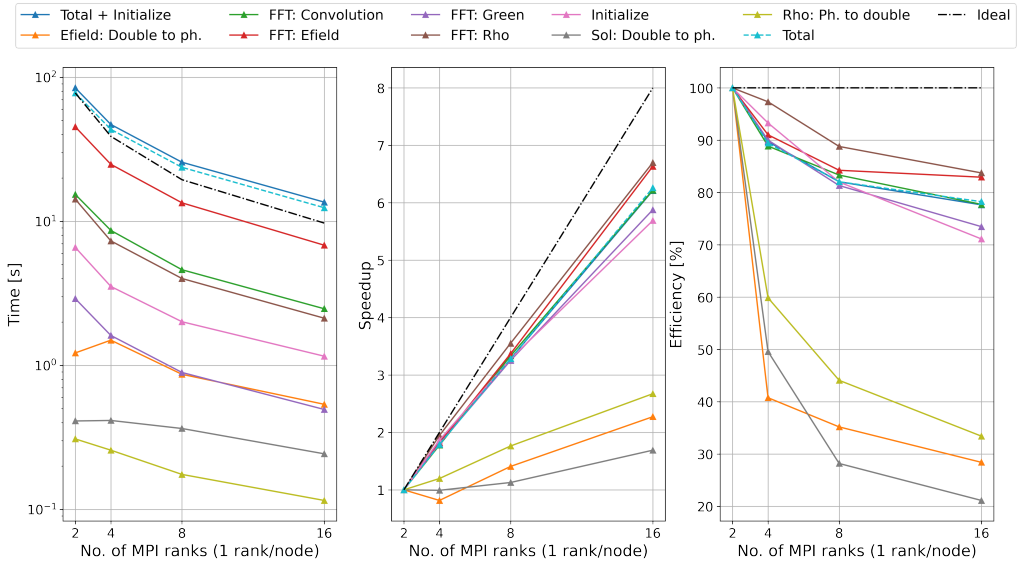

Figure B.4: Strong scaling test for pencil decomposition, point-to-point communication, with reordering.

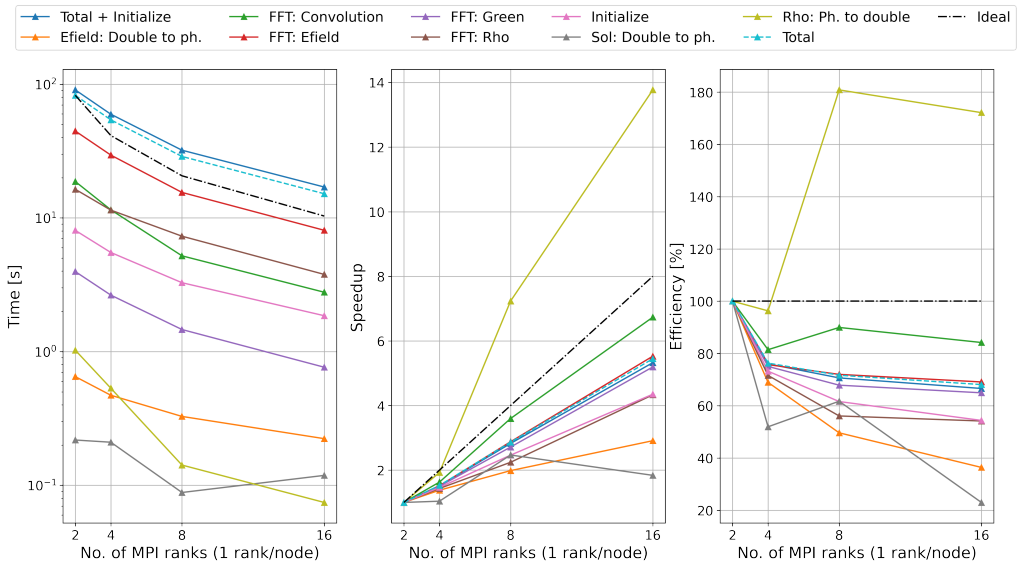

Figure B.5: Strong scaling test for slab decomposition, all-to-all communication, no reordering.

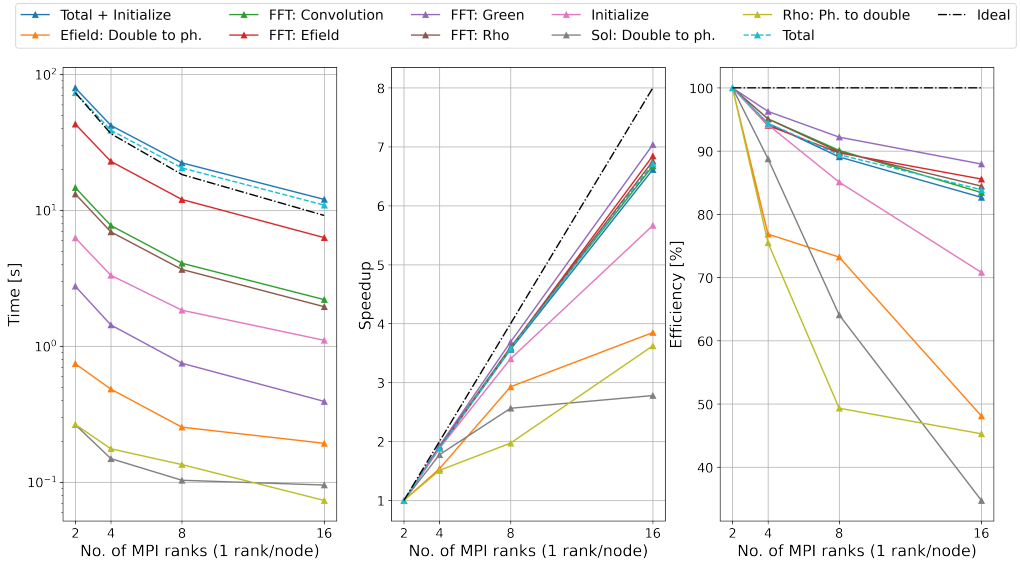

Figure B.6: Strong scaling test for slab decomposition, all-to-all communication, with reordering.

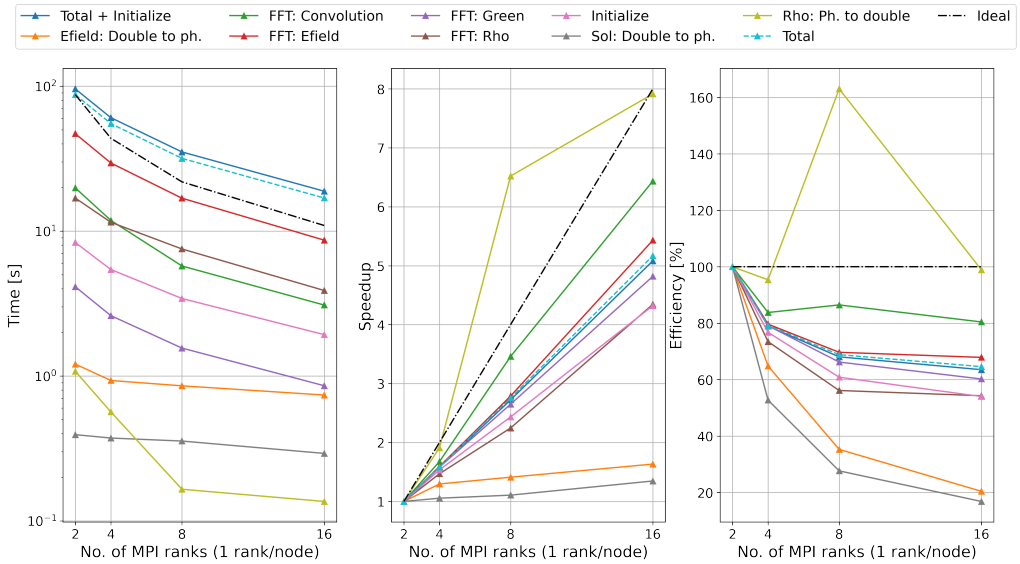

Figure B.7: Strong scaling test for slab decomposition, point-to-point communication, no reordering.

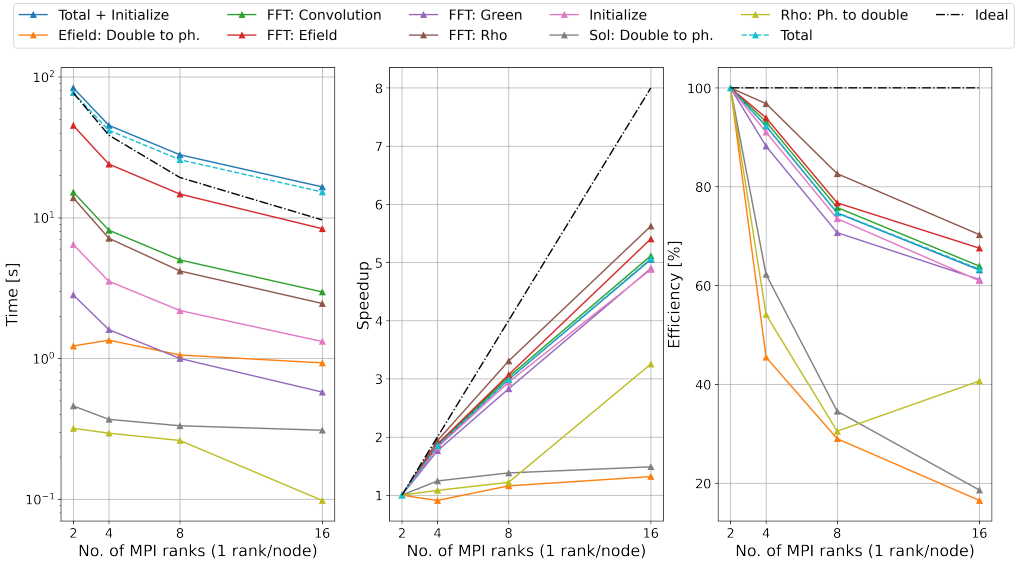

Figure B.8: Strong scaling test for slab decomposition, point-to-point communication, with reordering.

Appendix C: Scaling tests on GPU, 256^3

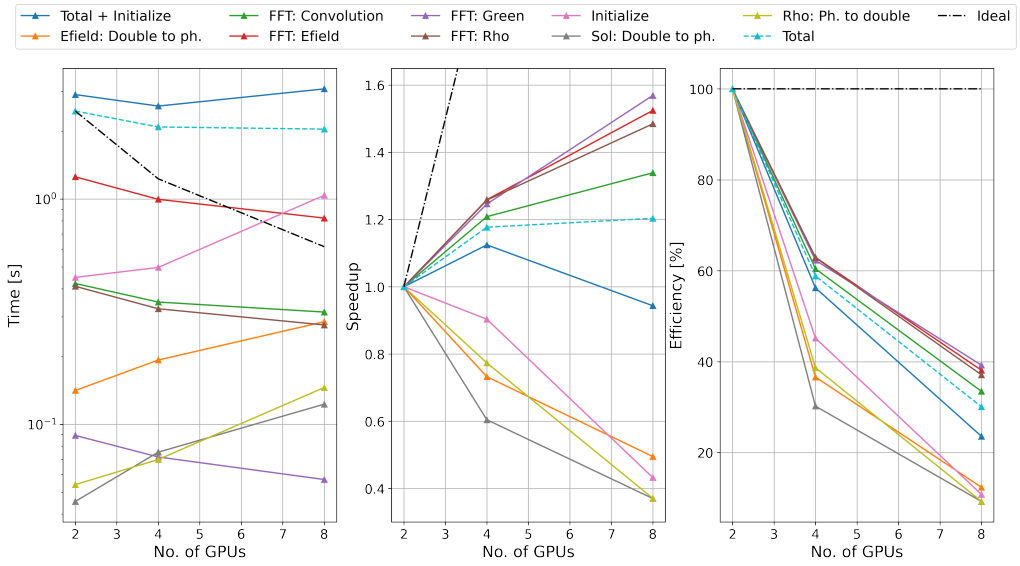

Figure C.1: Strong scaling test for pencil decomposition, all-to-all communication, no reordering.

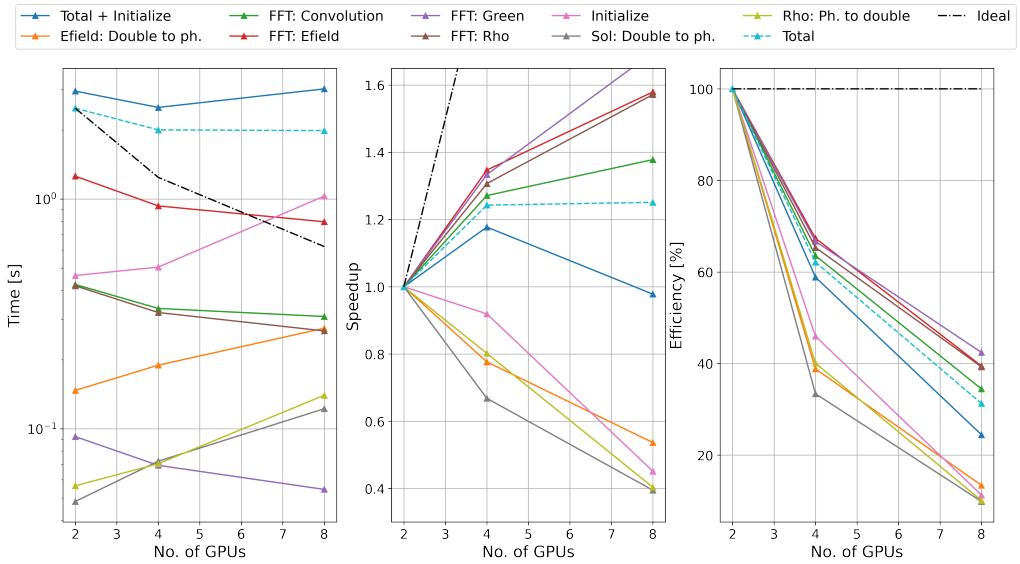

Figure C.2: Strong scaling test for pencil decomposition, all-to-all communication, with reordering.

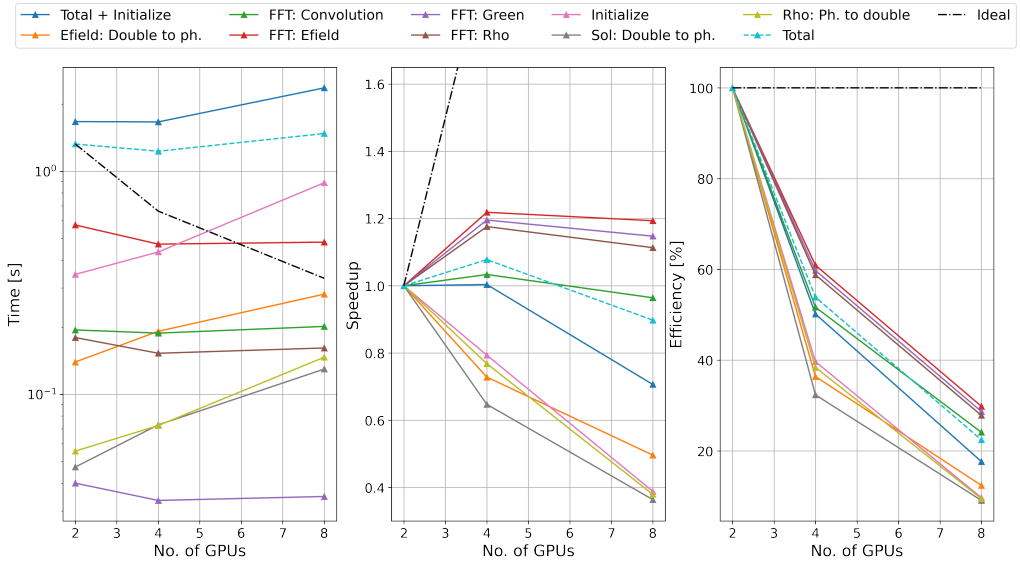

Figure C.3: Strong scaling test for pencil decomposition, point-to-point communication, no reordering.

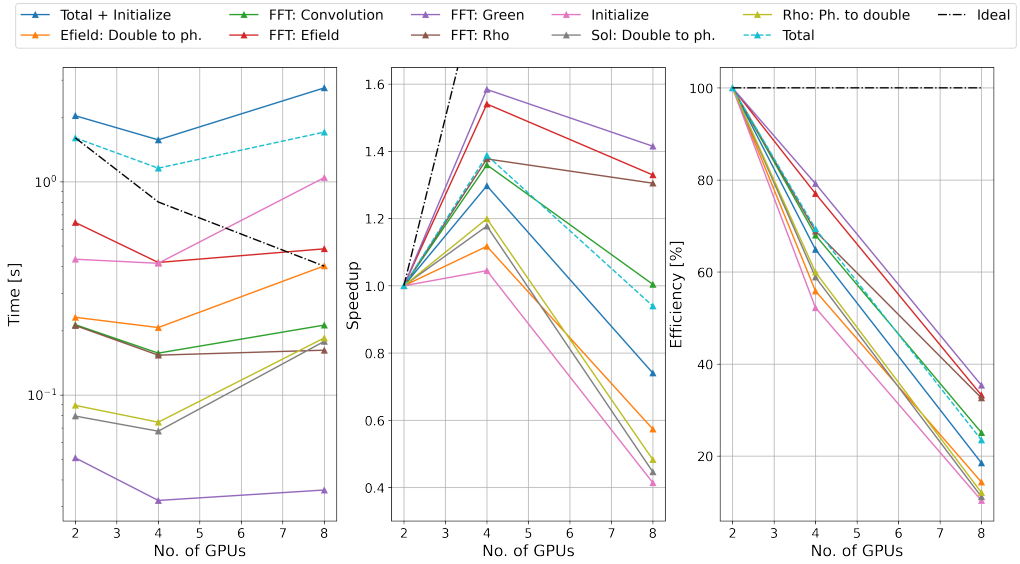

Figure C.4: Strong scaling test for pencil decomposition, point-to-point communication, with reordering.

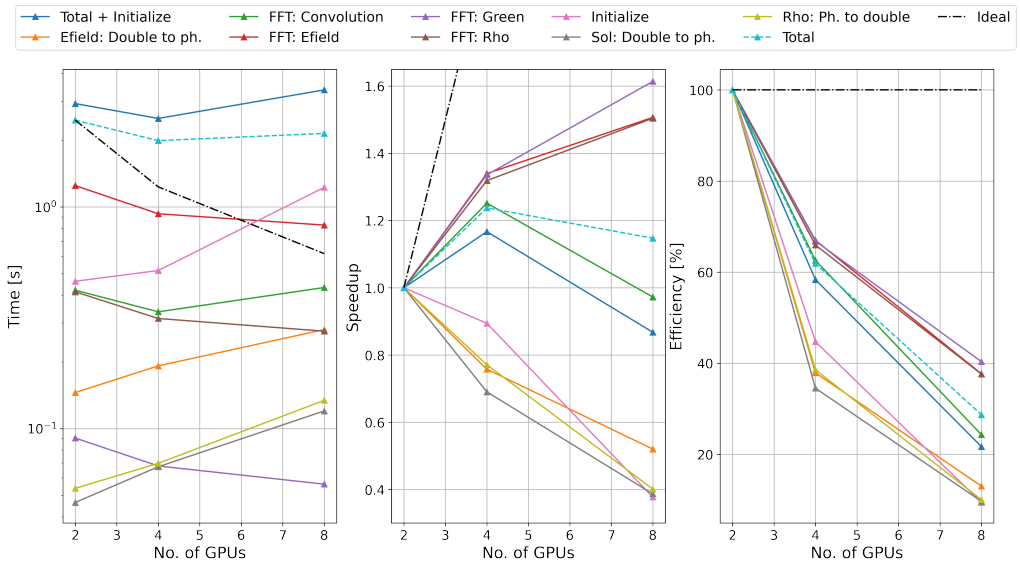

Figure C.5: Strong scaling test for slab decomposition, all-to-all communication, no reordering.

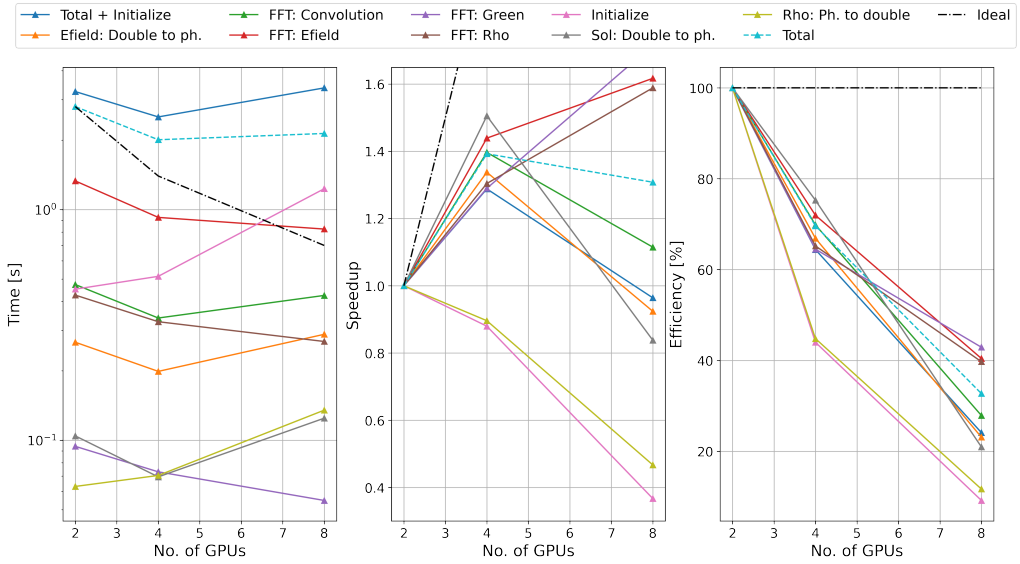

Figure C.6: Strong scaling test for slab decomposition, all-to-all communication, with reordering.

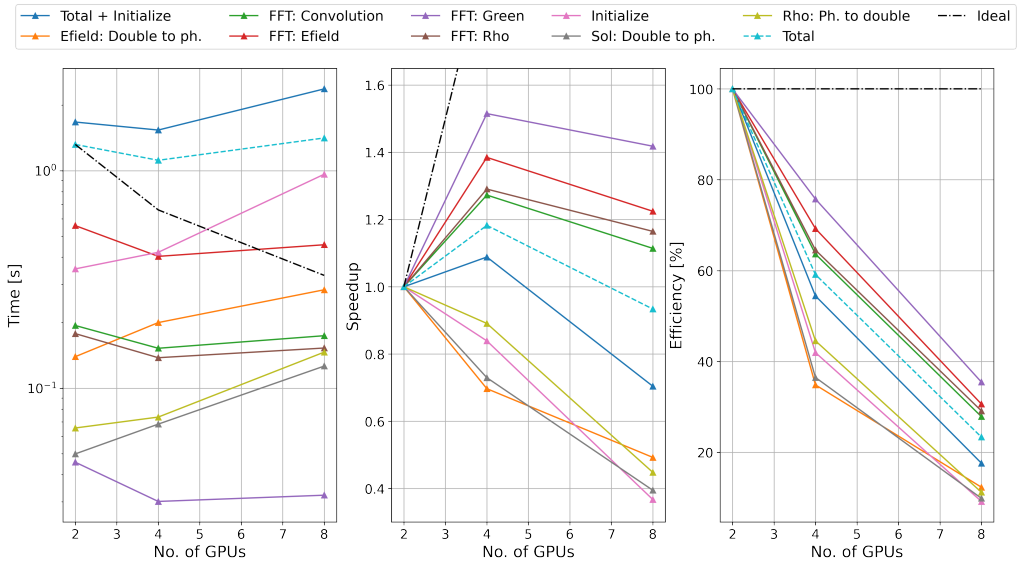

Figure C.7: Strong scaling test for slab decomposition, point-to-point communication, no reordering.

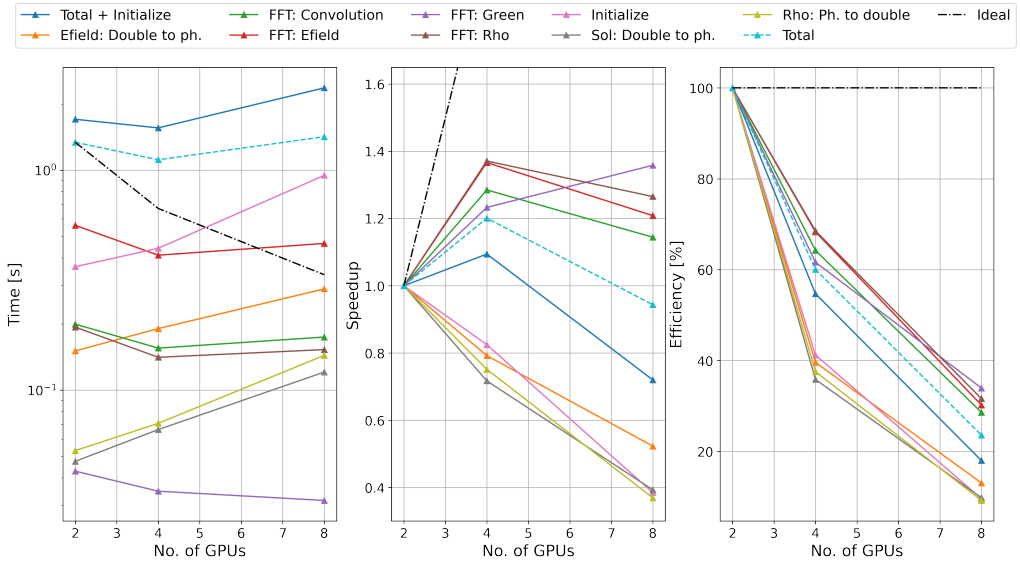

Figure C.8: Strong scaling test for slab decomposition, point-to-point communication, with reordering.

Appendix D: Scaling tests on GPU, 512^3

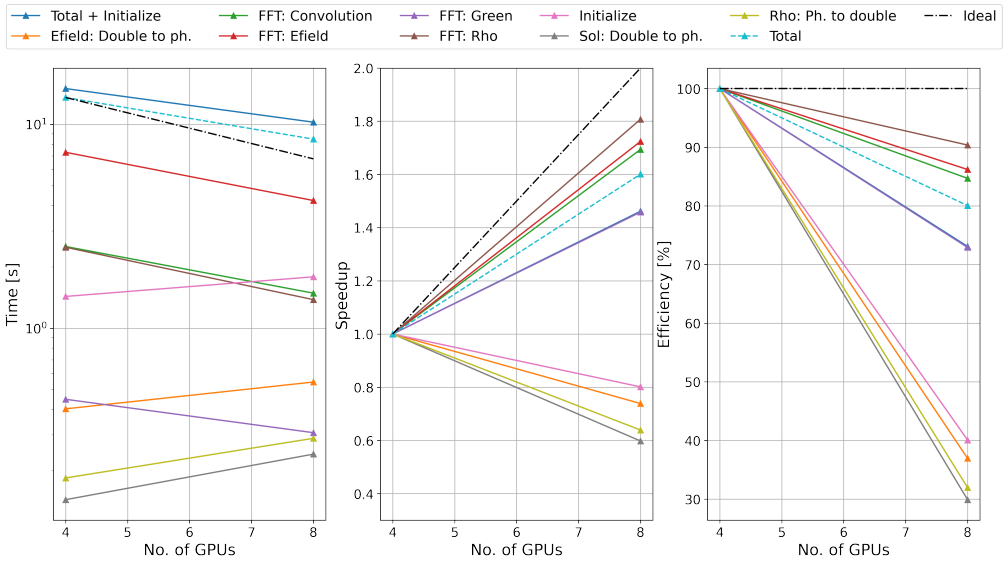

Figure D.1: Strong scaling test for pencil decomposition, all-to-all communication, no reordering.

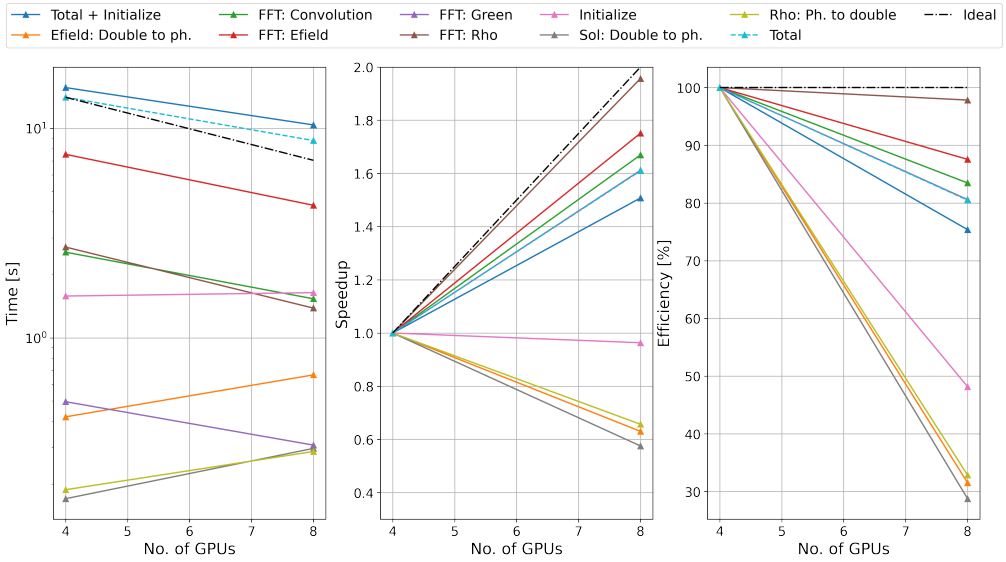

Figure D.2: Strong scaling test for pencil decomposition, all-to-all communication, with reordering.

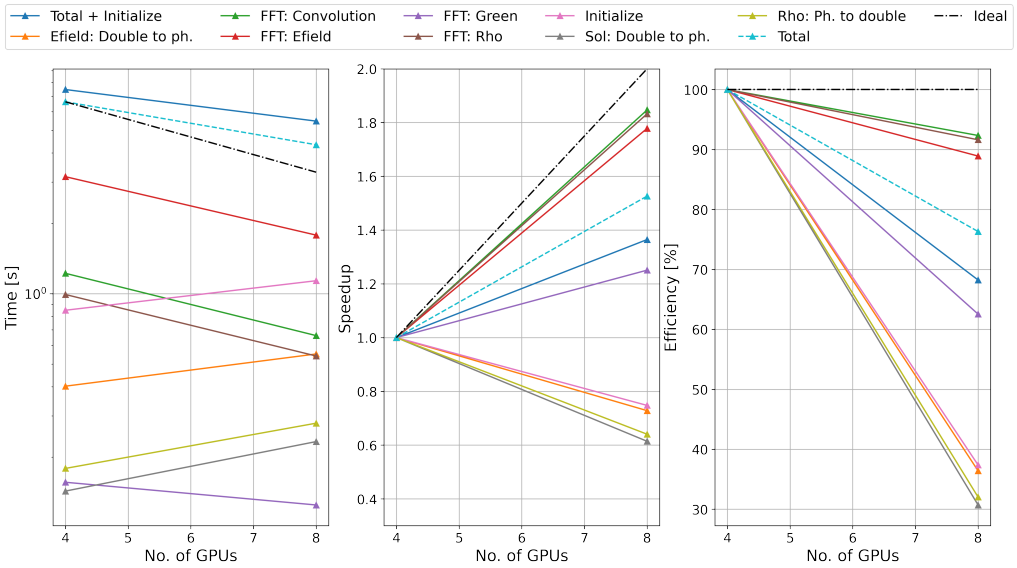

Figure D.3: Strong scaling test for pencil decomposition, point-to-point communication, no reordering.

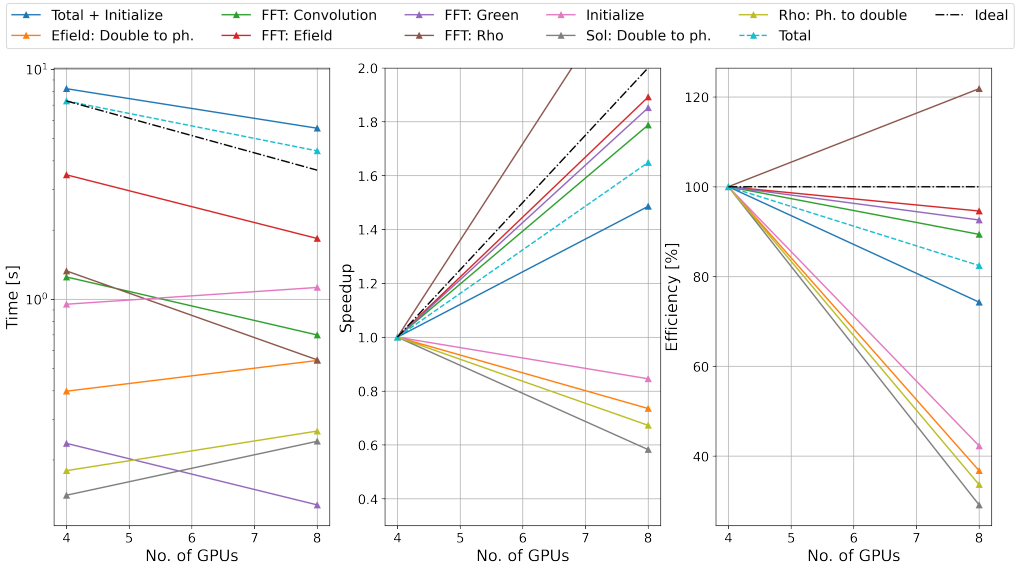

Figure D.4: Strong scaling test for pencil decomposition, point-to-point communication, with reordering.

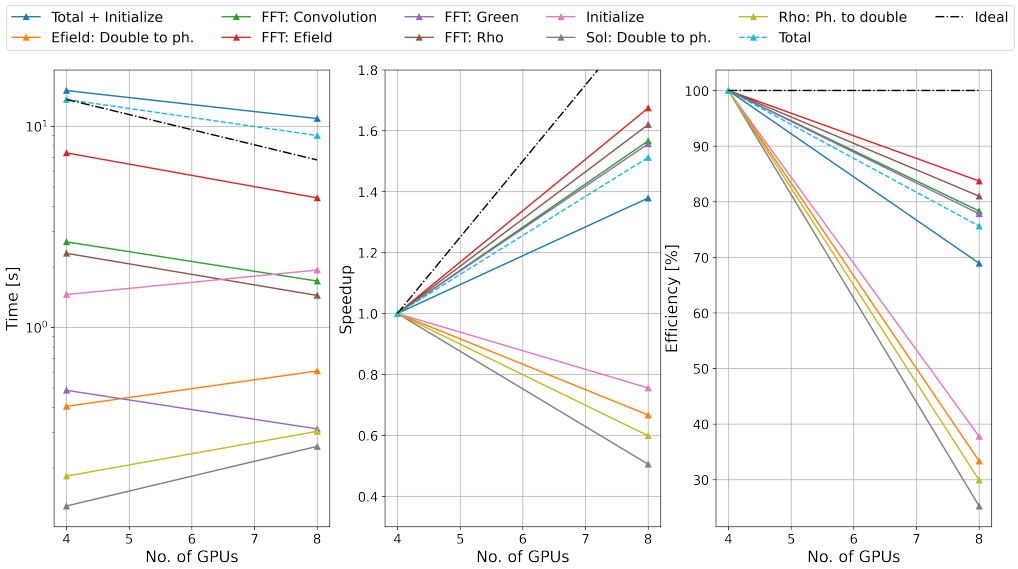

Figure D.5: Strong scaling test for slab decomposition, all-to-all communication, no reordering.

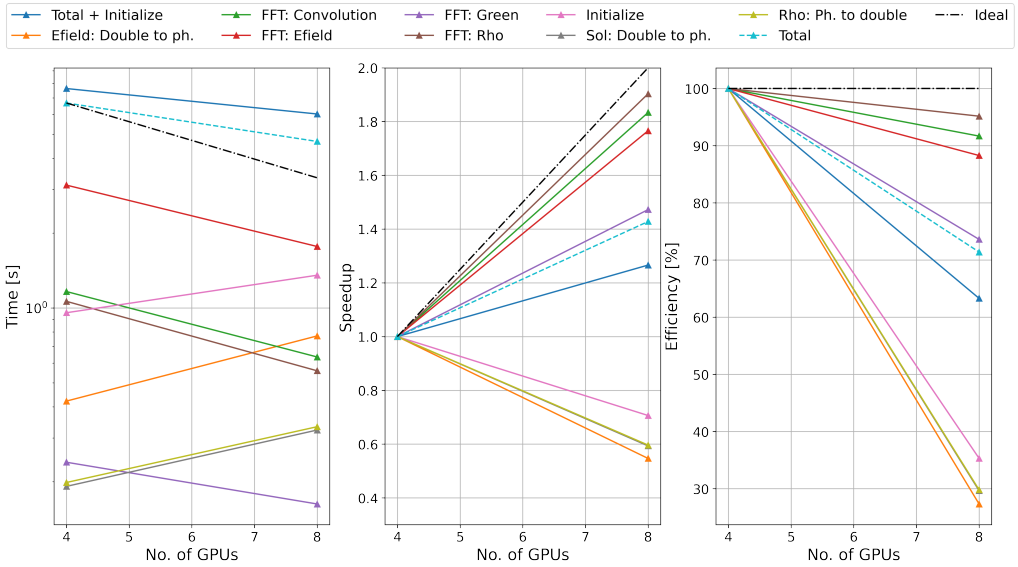

Figure D.6: Strong scaling test for slab decomposition, point-to-point communication, no reordering.