Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

# Implementing Zel'dovich Linear Initialization in the Performance Portable IPPL Library Cosmology Framework

Semester Project

in Computational Science and Engineering

Department of Maths

ETH Zurich

written by

Elisabeth Bobrova Blyumin

supervised by

Dr. A. Adelmann (ETH)

scientific advisers

Sonali Mayani

July 11, 2025

## Abstract

Cosmological $N$ body simulations begin by generating initial conditions (ICs) that represent the early universe's matter distribution. These ICs are typically Gaussian random fields drawing random Fourier modes that match the desired power spectrum. In this way, one obtains an initial density field that is statistically consistent with primordial fluctuations. To obtain particle positions and velocities, Lagrangian perturbation theory (LPT) is applied. The simplest approach is the Zel'dovich approximation (ZA) – a first-order LPT – which displaces particles from a uniform grid along the gradient of the initial potential. This is the approach taken in this semester project, of which the focus is to test the physicality of an initial conditions generator for the cosmology mini app in the Independent Parallel Particle Layer (IPPL) c++ framework. The bulk of this project focuses on checking the physics of the generated Gaussian field by means of a hermiticity test function.

# Contents

# List of Figures

# List of Tables

# List of Listings

# Chapter 1

# Introduction

The IPPL (Independent Parallel Particle Layer) library provides a performance-portable particle-mesh framework that runs unchanged on CPUs and GPUs by combining MPI for domain decomposition and Kokkos for node-level parallelism [1]. This library was primarily developed for the purpose of running simulations of particle-particle interactions governed by electromagnetic forces for plasma physics applications.

The IPPL "cosmo mini-app" has adapted the IPPL standard framework to also be used for large scale cosmological simulations [2]. The current cosmo mini-app follows the collision-less evolution of cold-dark-matter (CDM) particles in an expanding Friedmann–Lemaître–Robertson–Walker (FLRW) space-time. That prototype, however, still read its initial particle mesh with defined initial positions and velocity from external files produced by the N-GenIC initial conditions code which is publicly available [1].

The present project begins to remove this external dependency by embedding a self-contained Zeldovich initial condition (IC) generator in the IPPL cosmology code base. Embedding the Zel'dovich initial-conditions generator directly in the IPPL cosmology mini-app provides several advantages. First, generating the particles ourselves eliminates the IC files produced by external tools and packages. For a $512^3$ run this removes up to 20h of runtime when running on 8 CPU ranks with no multithreading. Even if the file only needs to be generated once, it still has to be stored and read in every time the simulation is run. In the implementation where IC are read in from an external file generated by NGENIC, the read in time accounts for more than 60% of the simulation when running the app for $512^3$ particles on 8 GPUs [2]. When creating the initial conditions from scratch, the generator uses the same domain decomposition as the solver, including HeFFTe for FFTs, and it should therefore inherit strong scaling behaviour improving the scalability of this step. Finally, including a native IC generator simplifies the workflow as it is reduced to a single Slurm batch script when running on a cluster, rather than needing multiple scripts to generate and recompile files into a single input file first, storing the file, and reading it in every time. Overall, this improves the performance portability of the programme as with a single script you will be able to run the code in serial, on CPU and on GPU without having to change anything except for the expected input file and command line arguments consisting of the physics specifications and desired hardware set-up.

The primary objective of this project is to verify the physical validity of the IC generator employed by the IPPL-based cosmology mini-app, rather than to maximise performance just yet. The Gaussian random field produced in Fourier space must obey Hermitian symmetry, $\delta(k) = \delta^*(-k)$, so that it's real-space counterpart is strictly real. Section 3 therefore introduces

[1]`https://www.h-its.org/2014/11/05/ngenic-code/`

a parallel hermiticity test. Given that the generated field can be domain decomposed when running on CPU, the -k partner of a mode often resides on a different MPI rank in a multi-rank setup, making cross-rank communication a necessity. The resulting exchange and its influence on scaling is analysed in Section 4.3.

The remainder of the thesis is organised as follows. Chapter 2 reviews the cosmological background and the Zel'dovich approximation on which the IC generator rests. Chapter 3 describes the implementation: Section 3.1 introduces the IC pipeline, and Section 3.2 details the hermiticity check in both single-rank and multi-rank form. Chapter 4 presents the results, beginning with validation of the hermiticity routine and a cosmological test implementation, and continuing with a scaling study that covers strong and weak scaling on CPUs and GPUs together with HPC-specific remedies. Chapter 5 discusses the physical assumptions, performance-portability aspects and observed scaling behaviour, and identifies avenues for code improvement. Chapter 6 concludes the thesis and outlines future directions while the appendices provide support for running the simulations to replicate the results of this project.

# Chapter 2

# Theoretical Background

The cosmo mini-app developed for IPPL models structure formation follows a model which is pressure-less, collision-less "dust" made entirely of cold dark-matter particles - the $\lambda$CDM model [2]. Radiation and baryonic physics are ignored, and dark energy enters only through the background expansion $a(t)$. Within this framework the cosmic evolution reduces to solving Newtonian $N$-body dynamics in co-moving coordinates. For an in-depth description of the full cosmological framework of the structure formation in this mini-app, please refer to [2]. The remainder of this chapter concentrates on how the initial conditions are generated and validated, beginning with the construction of a Gaussian random field that obeys the Zel'dovich approximation.

The Zel'dovich initial-condition procedure starts by drawing a Gaussian random density field. The displacement field $\mathbf{\Psi}(\mathbf{q})$ is related to the Fourier space density $\delta(\mathbf{k})$ by

$$\mathbf{\Psi}(\mathbf{k}) = -i\,\frac{\mathbf{k}}{k^2}\,\delta(\mathbf{k}), \qquad \delta(\mathbf{k}) = \sqrt{P(k)}\,(G_1 + iG_2), \tag{2.1}$$

where $G_{1,2}$ are independent gaussian random variables with mean 0 and variance 1 and $P(k)$ is the power spectrum which describes the initial distribution of matter and energy in the universe. The fourier space density field is hermitian, $\delta(-\mathbf{k}) = \delta^*(\mathbf{k})$, which guarantees a real values for the position and velocity after obtaining $\Psi(\mathbf{q})$ through an inverse FFT. At time $t$ the Eulerian position and velocity are

$$\mathbf{x}(t) = \mathbf{q} + D_1(t)\,\mathbf{\Psi}(\mathbf{q}), \quad \mathbf{v}(t) = \dot{D}_1(t)\,\mathbf{\Psi}(\mathbf{q}), \tag{2.2}$$

with $D_1 \propto a(t)$ being the linear growth factor [3]. Structure formation comes from tiny Gaussian density perturbations seen during the inflation period. The initial particle displacements are obtained from a Gaussian random field consistent with the target matter power spectrum $P(k)$.

The first–order Zel'dovich approximation (ZA) is the simplest Lagrangian-perturbation scheme and it suffices to test that our IC generator draws the correct Gaussian field and that the distributed hermiticity check succeeds across all ranks. However, modern literature documents the inaccuracies of ZA at later times and the transient modes it introduces [4], as well as the significant improvements obtained with second–order Lagrangian perturbation theory (2LPT) [5]. Incorporating those higher-order corrections is beyond the scope of the present exercise, but is recommended as a next step.

# Chapter 3

# Methods

## 3.1 Initial Conditions Generation

The cosmological simulation presented in this work implements a performance-portable initialization of the large-scale cosmological structure using the Zel'dovich approximation, directly within the cosmo mini-app simulation framework based on IPPL. This is added in addition to the previous approach of reading externally generated initial conditions, namely those from NGenIC.

The framework was updated such that an input file specifies cosmological parameters (whereas previously these parameters were defined either in command line arguments, or were hard coded) and a flag `ReadInParticles=0 | 1`. Setting it to 0 now triggers the built-in generator, whereas keeping `ReadInParticles = 1` will enable the app to read the ICs from file. The method to run the code by reading in an existing IC file is outlined in Appendix B.

When `ReadInParticles = 0`, the function `LinearZeldoInitMP` is called to generate the initial conditions. The initial conditions for particle positions and velocities are constructed using the Zel'dovich approximation, which provides a linear mapping from initial Gaussian density perturbations to particle displacements and velocities. First, the fourier space density field $\delta(k)$ is initialised as a field with Gaussian random modes. In the code,

- `cfield_m` stores $\delta(\mathbf{k})$ and is later reused to store the three displacement components,
- `Pk_m` stores $P(k)$ (during debugging we set $P = 1$, but the power spectrum can also either be generated on the fly or given as an input file).

A Kokkos `parallel_for` iterates over local plus ghost cells. The local index of the cell is converted to a global index. For each global index $(i, j, k)$ the corresponding negative partner $(N_x - i, N_y - j, N_z - k)$ is computed. To generate a Gaussian random field that satisfies the Hermitian symmetry condition $\delta(-\mathbf{k}) = \delta^*(\mathbf{k})$, we deterministically assign complex Fourier amplitudes to each grid point using a globally seeded random number generator. This approach ensures that both a mode $\mathbf{k}$ and its conjugate partner $-\mathbf{k}$ are always assigned consistent values without requiring any cross-rank communication.

As stated above, for each global Fourier index $(i, j, k)$, the Hermitian-conjugate index is defined as

$$(i_{\mathrm{neg}}, j_{\mathrm{neg}}, k_{\mathrm{neg}}) = (N_x - i,\ N_y - j,\ N_z - k), \tag{3.1}$$

with care taken to preserve symmetry when $i$, $j$, or $k$ are zero. It is then determined whether the current mode is its own conjugate (as a Nyquist or zero mode), or if it is lexicographically smaller than its negative counterpart. Only the lexicographically smaller of each Hermitian pair proceeds to generate the random numbers.

From this "owner" index, a unique key is computed

$$\texttt{key\_index} = (i_{\text{key}} \cdot N_y + j_{\text{key}}) \cdot N_z + k_{\text{key}}, \tag{3.2}$$

and the XOR operation is applied with a fixed global seed to create a 64-bit integer $x$. This integer is then passed through two rounds of the XorShift64 pseudorandom number generator:

$$x \leftarrow x \oplus (x \ll 13); \quad x \leftarrow x \oplus (x \gg 7); \quad x \leftarrow x \oplus (x \ll 17), \tag{3.3}$$

to yield two independent 64-bit outputs $r_1$ and $r_2$. The 53 most significant bits are extracted from each result and normalized to the unit interval:

$$u_{1,2} = \frac{r_{1,2} \gg 11}{2^{53}}, \tag{3.4}$$

producing two uniform random numbers in $[0, 1)$. These are transformed into standard Gaussian modes using the Box-Muller method:

$$R = \sqrt{-2 \ln u_1}, \tag{3.5}$$

$$\theta = 2\pi u_2, \tag{3.6}$$

$$G_{\text{re}} = R \cos\theta, \tag{3.7}$$

$$G_{\text{im}} = R \sin\theta. \tag{3.8}$$

These random values are scaled by the square root of the linear power spectrum $P(k)$ to form the final complex amplitude:

$$\delta(\mathbf{k}) = \begin{cases} \sqrt{P(k)} \cdot G_{\text{re}}, & \text{if self-conjugate (imaginary part zero)} \\ \sqrt{P(k)/2} \cdot (G_{\text{re}} + iG_{\text{im}}), & \text{if owner of pair} \\ \sqrt{P(k)/2} \cdot (G_{\text{re}} - iG_{\text{im}}), & \text{if partner of pair.} \end{cases} \tag{3.9}$$

This construction guarantees that the Hermitian symmetry is satisfied exactly and without requiring explicit communication between devices or MPI ranks. The key benefit of this approach is that the random field is fully reproducible and consistent across architectures, while also being performance-portable (requiring no change for different hardware specifications).

After constructing $\delta(\mathbf{k})$, a verification step checks that Hermitian symmetry indeed holds across all ranks as described in Section 3.2. For each Fourier mode $\mathbf{k}$, the code checks that $\delta^*(\mathbf{k}) = \delta(-\mathbf{k})$ within the tolerance of the double value of $\delta(k)$. Any detected violation is flagged in the error logs. The check is done via a boolean function `isHermitian()`.

Then, the displacement field is computed component-wise in Fourier space using:

$$\Psi_d(\mathbf{k}) = i\frac{k_d}{k^2}\delta(\mathbf{k}). \tag{3.10}$$

A backward FFT (using IPPL's HeFFTe wrapper) is applied to obtain $\Psi_d(\vec{q})$ on the grid. For each spatial component $d \in \{x, y, z\}$, the kernel multiplies $\delta(\mathbf{k})$ by $ik_d/k^2$ (skipping the $k = 0$ mode), takes an inverse FFT, and writes the real part into `cfield_m`. A final loop perturbs the positions of the particles from the cell centers $\mathbf{q}$ to $\mathbf{x} = \mathbf{q} + \Psi$ and stores the corresponding velocities.

The particles are initialized on a uniform grid. For each particle, the displacement in each dimension is interpolated from the corresponding $\Psi_d(\vec{q})$. The positions are then updated as $x_d = q_d + D_1\Psi_d(q)$, and the velocities are set as $v_d = \dot{D}_1\Psi_d(q)$, as by Equation 2.2, with the linear growth factors $D_1, \dot{D}_1$ set to 1.

This approach ensures that we have performance portability via Kokkos and IPPL abstractions, supporting CPU and GPU backends and physical correctness due to Hermitian enforcement. Finally, we also get speed and reproducibility as no I/O is required for initial conditions. By integrating the Zel'dovich initialization directly into the simulation, the framework is sped up by skipping the read in time and it gains full control over cosmological parameters without external preprocessing steps.

## 3.2 Hermiticity Check Function

`isHermitian()` validates that the Fourier Space density field $\delta(\mathbf{k})$ obeys symmetry: $\delta(-\mathbf{k}) = \delta^*(\mathbf{k})$.

The function first checks the total number of MPI ranks and then executes either a single-rank or a multi-rank path. Throughout the discussion, the local index on rank $r$ is denoted by $(i, j, k)$, its Hermitian partner index by $(-i, -j, -k)$, and the global grid extents by $(N_x, N_y, N_z)$. Machine precision is obtained once via `tol = std::numeric_limits<double>::epsilon()` and is reused in every comparison, avoiding both floating precision errors that result in a difference greater than 0 and arbitrary user chosen thresholds.

### 3.2.1 Single-rank

The structure of the single rank hermiticity test can be seen in Listing 3.2.1. When `nranks == 1` the check is performed on device (there may be multiple threads). A three–dimensional Kokkos `parallel_reduce` iterates over the full domain and evaluates

$$\left|\mathrm{Re}\,\delta(-\mathbf{k}) - \mathrm{Re}\,\delta^*(\mathbf{k})\right| \le \varepsilon \quad \wedge \quad \left|\mathrm{Im}\,\delta(-\mathbf{k}) + \mathrm{Im}\,\delta(\mathbf{k})\right| \le \varepsilon, \tag{3.11}$$

where $\varepsilon$ is the tolerance specified earlier. The reduction uses `Kokkos::Min<int>` to reduce the flag to the lowest of two values - 0 or 1. Any hermiticity violation flips the flag to 0, and the overall `bool isHermitian()` function hence returns `false`.

Listing 3.1: Pseudocode for the single-rank Hermiticity check

```
function isHermitian()
    tol          // machine_epsilon
    myrank       // MPI_Rank()
    nranks       // MPI_Size()
    (Nx,Ny,Nz) // global grid sizes

    if nranks = 1 then
        parallel for each k do
            kneg // partnerOf(k)
            if |field(kneg) - conj(field(k))| > tol then
                localFlag // 0
            end if
        end for
        return (localFlag != 0)
    end if
```

### 3.2.2 Multi-rank

The pseudo code in Listing 3.2.2 illustrates the case where multiple MPI ranks are used for the hermiticity test. For `nranks > 1` the negative partner of a $\delta(\mathbf{k})$ fourier mode may lie on a different MPI rank, so the logic is split into four phases to facilitate communication.

**(1) Local scan and send count (lines 9-21).** First, we loop over all local indices. For every $(i, j, k)$ the owner of $(-i, -j, -k)$ is found by checking the domain limits of each domain using `layout.getDeviceLocalDomains()`. Three outcomes are possible:

1. $-\mathbf{k}$ is owned by the same rank $r$ and an immediate hermiticity check is performed on the pair.
2. $-\mathbf{k}$ is owned by a different rank, $q \neq r$, and the array that counts how many items must be sent to each rank, denoted $c_r$, is atomically incremented at the relevant index (using `atomic_fetch_add` to ensure that two threads cannot write the same variable at the same time).
3. No owner of $-\mathbf{k}$ is found (partitioning error) and the local flag is set to 0 and an error message is printed.

**(2) Host prefix scan (lines 23-25).** The send count array $c_r$ for rank $r$, is initially filled on device. However, for MPI cross-rank communication information, such as displacements and buffer sizes, must be constructed on the host, so this array is copied to host memory. To prepare a flat communication buffer that holds all outgoing messages, we must compute a displacement array $s_r$, which gives the starting index of rank $r$'s data within the contiguous send buffer. This is achieved by performing a prefix sum over the count array $c_r$. The total number of items to be sent is then given by

$$n_{\text{send}} = s_{n_{\text{ranks}}-1} + c_{n_{\text{ranks}}-1} \tag{3.12}$$

This total defines the size of the flat buffer. The displacement array ensures that data from different ranks are written into non-overlapping contiguous regions. To illustrate, consider an example with 4 ranks, where the count array is

$$[c_0, c_1, c_2, c_3] = [3,\ 2,\ 4,\ 1]$$

Computing the prefix sum gives:

$$[s_0, s_1, s_2, s_3] = [0,\ 3,\ 5,\ 9]$$

The flat send buffer will then be indexed as follows:

```
Index : 0  1  2  3  4  5  6  7  8  9
Buffer: [R0 R0 R0 R1 R1 R2 R2 R2 R2 R3]
```

Each rank's data begins at its corresponding displacement $s_r$ and occupies $c_r$ entries. For example, rank 1's data starts at index $s_1 = 3$ and occupies $c_1 = 2$ entries, indices 3 and 4. This layout ensures that the communication buffers can be constructed and accessed correctly during an `MPI_Isend`. Since MPI communication naturally happens on host, both $c_r$ and $s_r$ must reside in host memory.

Given that every $+\mathbf{k}$ sent by rank $r$ is the $-\mathbf{k}$ needed by exactly one other rank, the number of receives is identical to the number of sends and the same displacement array is reused for posting `MPI_Irecv`. Finally, we copy the displacements per destination rank back to device memory as the packing is done in parallel using a kokkos loop.

Listing 3.2: Pseudocode for the distributed Hermiticity check

```
function isHermitian()
    tol         // machine_epsilon
    myrank      // MPI_Rank()
    nranks      // MPI_Size()
    (Nx,Ny,Nz) // global grid sizes

    if nranks > 1 then
        /* --- Pass 1: count remote pairs & test local ones --- */
        parallel for each local (i,j,k) do
            if (i,j,k) = (0,0,0) then continue //skip DC mode
            kneg  // (Nx-i, Ny-j, Nz-k)
            owner // rankThatOwns(kneg)

            if owner = myrank then             // partner is local
                if |field(kneg) - conj(field(i,j,k))| > tol then
                    localFlag // 0
                end if
            else                               // partner is remote
                atomic_fetch_add(sendCount[owner],1)
            end if
        end for

        /* allocate device buffers using sendCount prefix sums */
        compute send_disp[ ] and total_sends
        allocate sendBuffer[total_sends], recvBuffer[total_sends]

        /* --- Pass 2: pack messages --- */
        parallel for each local (i,j,k) do
            kneg  // (Nx-i,Ny-j,Nz-k)
            owner // rankThatOwns(kneg)
            if owner != myrank then
                slot // send_disp[owner] +
                       atomic_fetch_add(sendCount[owner],1)
                sendBuffer[slot] // (i,j,k, field(i,j,k))
            end if
        end for

        /* GPU-direct non-blocking exchange */
        post Irecv(recvBuffer), Isend(sendBuffer)
        MPI_Waitall()

        /* --- Pass 3: verify remote pairs --- */
        parallel for idx = 0 .. total_sends-1 do
            p            // recvBuffer[idx]
            kneg_local   // partnerOf(p.k)
            if |field(kneg_local) - conj(p.value)| > tol then
                localFlag // 0
            end if
        end for

        globalFlag // MPI_Allreduce(MIN, localFlag)
        return (globalFlag != 0)
```

**(3) Packing on the device (lines 27-36).** Once the total size and displacements $s_r$ of the outgoing messages have been determined, the flat send buffer on the device can be filled with actual data. This packing is done in parallel, using a second loop over the local grid. Each thread is responsible for checking whether the $-\mathbf{k}$ partner of its assigned Fourier mode $\mathbf{k}$ resides on a different rank. If so, it packs the data corresponding to mode $\mathbf{k}$ into a shared linear buffer. Each message contains 5 pieces of data - the 3 $\delta(k)$ wavevector components and the 2 Fourier amplitude real and complex components.

To ensure that each message occupies a unique slot, we need to add the number of messages being sent to the displacement index, as described previously.

$$\text{slot} = s_r + c_r, \tag{3.13}$$

where $s_r$ is the start index (displacement) of messages destined for rank $r$, and $c_r$ is a per-destination counter that tracks how many items have already been placed in that section. The atomic add guarantees that each thread obtains a unique offset within its rank's segment of the buffer. To store each message, a struct `HermitianPkg` was defined at the top of the header file containing (`int kx, ky, kz` and `double re, im`). Each slot in the buffer is of type `HermitianPkg`, meaning that all five fields required for the Hermitian check are already allocated together in memory. This removes the need for multiple arrays or multidimensional data structures, and allows us to use a flat buffer with simple indexing.

This approach has several advantages. First, the flat buffer enables us to pack the message into a single, contiguous array for MPI communication, preventing additional communication overhead in an already cross-rank communication-heavy task. Then, by computing and storing displacements $s_r$ in advance, the layout is predetermined and we avoid dynamic memory allocation at communication time. The use of atomic adds for counting and indexing avoids write collisions, while enabling fully parallel packing on the device. Finally, the `HermitianPkg` struct allows all required data to be stored and transmitted as a single element, simplifying the indexing calculations. The result is a single flat buffer array that contains all outgoing messages ready for MPI communication.

**(4) Communication and final check (lines 38-52).** All receives are posted first directly into contiguous regions of `recv_buffer_d`. Next, the sends are issued with the same tag (0) because source/destination/order are already disambiguated by the layout of the buffers, and the order in which they arrive at each rank is irrelevant. `MPI_Isend` and `MPI_recv` are used for communication. They are non-blocking sends, which allows for all the messages to be sent without waiting for each rank to post the corresponding receives, and thereby prevents deadlocking [6]. These calls expect the type of each message being sent to be specified to enable successful memory allocation. In this case, each element of the buffer array that is being sent is a struct which mixes int and double types. Therefore, the types `MPI_DOUBLE` or `MPI_INT` cannot be used as the data type specified in communication. Instead, `MPI_BYTE` is used and the number of bytes is specified to match `sizeof(HermitianPkg)`. After `MPI_Waitall` and a device `fence`, which are there to make sure all communication is finished before proceeding, a final parallel loop iterates over the received packages and verifies Hermitian symmetry exactly as described in the single-rank case (Section 3.2.1).

A final integer reduction (`MPI_Allreduce(min)`) is performed on each rank's local variable `localHermitian`. The return statement of the `isHermitian()` function is `localHermitian != 0`, returning `true` if all Fourier modes across all ranks satisfy the symmetry to machine precision $\varepsilon$.

**Summary** The `isHermitian()` function:

1. Chooses the single- or multi-rank algorithm at run time.
2. In the multi-rank case, packs the send buffer and all the messages and communicates the values using non-blocking MPI exchanges.
3. Compares complex pairs up to the numerical limit $\varepsilon$.
4. Returns a Boolean value indicating whether hermitian symmetry was upheld or not.

This guarantees that any subsequent inverse transform produces a purely real density field, ensuring physical validity of the generated Zel'dovich initial conditions.

# Chapter 4

# Results

All the scripts to reproduce the results can be found in the semester project gitlab[1], and compilation and running of the scripts is described in detail in Appendix A.

## 4.1 Hermiticity Function Validation

To test whether the function `isHermitian()` is a valid test of Hermitian symmetry, the two test cases below were developed:

1. a single-mode field in which only the modes at $\mathbf{k}_0$ and $-\mathbf{k}_0$ are populated
2. a full Gaussian field in which each mode amplitude is drawn from a complex Gaussian of variance $P(k) = 1$ and paired with its Hermitian conjugate (this random gaussian field is built identically as in the provided code in the `LinearZeldoInitMP()` in `StructureFormationManager.h`).

### 4.1.1 Algorithmic description

A unit test was created for the `isHermitian()` function, and can be found under `../ippl/test/cosmology/TestHermiticity.cpp` in the IPPL framework. In this test, a `StructureFormationManag` object is constructed and the `pre_run` method is called to assemble the grid and allocate the distributed complex field in Fourier space. Then the Kokkos view is defined as `cview`. This view gives access to the full field such that we can iterate over every complex mode. To do this, we need the global grid extents $(N_x, N_y, N_z)$ and the ghost cells $ngh$ so to later translate local indices into global wavenumbers. These values are accessed using get methods that are only available if the code is compiled with the test flag on (see Appendix A.1 and A.3) on steps outlining how to compile the code and run the test).

**Test 1 - a single-mode field.** To construct an input that satisfies Hermitian symmetry, the test first clears the entire array that was previously initialised in the StructureFormationManager class to zero on all ranks and then writes the same real value $0.5 + 0.0, \mathrm{i}$ into the two modes at $\mathbf{k}_0 = (2, 1, 0)$ and $-\mathbf{k}_0$. Calling `isHermitian()` now performs a parallel reduction over all ranks and is expected to return `true`. The field is then reset to zero once more and immediately rewritten so that the positive mode still carries $0.5 + 0.0, \mathrm{i}$ but its partner is deliberately assigned

[1]`https://gitlab.ethz.ch/ebobrova/semester-project`

$-1.0 + 0.0, \mathrm{i}$, thereby destroying the conjugate relation. Running `isHermitian()` again should therefore yield `false`. With these two steps we verify that the function accepts a genuinely Hermitian configuration and rejects an almost identical one in which the symmetry has been violated.

**Test 2: full Gaussian field.** After re-initiating all the field values to zero once more, the code launches a second kernel that visits every index and draws two uniform pseudo-random numbers which are converted via the Box–Muller transform into a pair of independent Gaussian variates. For modes that are their own conjugate (DC and Nyquist planes) the imaginary part is forced to zero and the real part is scaled by $\sqrt{P(k)}$ with $P(k) = 1$. For all other modes only the lexicographically smaller of each $(\mathbf{k}, -\mathbf{k})$ pair is populated with $\sqrt{P(k)/2}, (G_{\mathrm{re}} + iG_{\mathrm{im}})$, and its partner is implicitly the complex conjugate, guaranteeing global Hermiticity. This approach is identical to the one described in Section 3.1. The call to `isHermitian()` must return `true` for such a randomly generated gaussian field. Finally, to confirm that the function also flags random fields that are non hermitian, the kernel revisits every non-self-conjugate pair and flips the sign of the imaginary part in the "conjugate" partner, thereby breaking hermitian symmetry. The final call to `isHermitian()` is expected to return `false`.

### 4.1.2 Results

Running on both GPU and CPU back-ends produced identical outcomes:

Listing 4.1: TestHermiticity.cpp output

```
hermiticityTest1> [1/4] ... TRUE
hermiticityTest1> [2/4] ... FALSE (expected)
hermiticityTest2> [3/4] ... TRUE
hermiticityTest2> [4/4] ... FALSE (expected)
```

Hence Listing 4.1 confirms that `isHermitian()` correctly identifies both symmetric and intentionally broken fields on single- and multi-rank runs. To view the raw results generated from these tests, refer to the output logs labeled `test*.out` and `test*.err` in `results/cosmo-new/` of the semester project gitlab[2]. To view the scripts used to generate these results see `job-scripts/run_test.sh` and `job-scrips/run-gpu_test.sh`. To run the tests yourself follow the steps in Appendix A.3.

## 4.2 Hermiticity Test

In the cosmology code, the function `LinearZeldoInitMP()` generates an initial condition (IC) density field as a complex Gaussian random field with target power spectrum $P(k)$. For the duration of this project the spectrum is fixed to unity, $P(k) = 1$, in order to isolate numerical issues from cosmological ones. As described previously, after the first kernel initialises the fourier density field with gaussian random modes, `isHermitian()` is called. If the field is truly hermitian, then the rank 0 will print a statement confirming hermiticity to the output log of the job, as shown by the output in Listing 4.2.

Listing 4.2: StructureFormation output snippet

```
LinearZeldoInitMP {0}> Fourier density field is Hermitian.
```

[2]`https://gitlab.ethz.ch/ebobrova/semester-project`

In this project, the cosmo-app simulation was run for 0 iteration steps (meaning, only generating the initial conditions) to check the success of the Hermiticity check. This was checked for on multiple CPUs in the merlin6 cluster ($n_r = 1, 2, 3, 4, 5$) and on up to 8 gpus on the gwendolen machine (both set-ups are part of the PSI high performance computing clusters). In all cases, the above line from Listing 4.2 was printed in the output logs, confirming both the functionality of the workflow and that the generated density field is hermitian.

The scripts used to run the simulation can be found under `job-scripts/run.sh` and `job-scripts/run-gpu.sh`, whereas the explanations on how to compile the code and run them are outlined in Appendix A. The output logs with the results that were generated showing successful generation of initial conditions and hermiticity test outputs are stored in `results/cosmo-new/cosmo-*.out`.

The successful result demonstrates that the IC generator obeys Hermitian symmetry even before the physical $P(k)$ is introduced. When the full spectrum is switched on, the same test will serve unchanged to ensure that the newly generated field preserves the symmetry.

## 4.3 Scaling Studies

### 4.3.1 Scaling set-up

Parallel scalability tests help to quantify the raw performance gain that additional hardware gives and they expose algorithmic bottlenecks such as communication, synchronisation and possible load imbalance. Because the Hermiticity check loops over every Fourier mode and performs several communications across ranks, its cost is expected to be communication–dominated for small problems where communication overheads will be very visible. The experiments below were implemented to compare that expectation with reality on CPU and on GPU hardware. Thereby, the goal of the scaling studies is most inline with the second purpose described - to expose the algorithmic bottlenecks and give suggestions for future implementations to improve performance.

**Architectures.** CPU scaling studies were done on the merlin6 cluster at PSI. GPU runs were performed on the gwendolen machine. Both builds used IPPL v3.2.0 and CUDA 12.2 was employed for the GPU runs, while OpenMP and OpenMPI 4.1 were used for CPU. To see the full specifications of the compiled IPPL in this section see Appendix A.1, and to Appendix A.4 to see how to run the scaling studies done in this section.

**Problem settings.** Table 4.1 summarises the numerical grid sizes and node counts explored in the two scenarios:

- Strong scaling - the grid size $N = N_x^3$ is fixed while the number of MPI ranks $P$ grows.
- Weak scaling - the local workload $N/P$ is fixed so that both $N$ and $P$ rise proportionally.

The strong scaling job scripts are simply a nested for loop where each problem size is run on every node/GPU set-up. The figures for the strong scaling contain 4 lines each, representing the different results per problem size. The largest problem size had $N_x = 128$, $N = 128^3 = 2,097,152$ particles in total. The larger problem sizes $N_x = 256, 512$ caused an out of bounds memory error on the merlin cluster setups. For the weak scaling, the numbers for the linear grid size might seem arbitrary at first glance. They were chosen in a way such that the total number of particles $N_x^3$ doubles at each increase in node/GPU count. These numbers are not whole numbers, so

Table 4.1: Parameters of the scaling studies. All nodes and GPUs were exclusive (each run had access to full memory of a node).

| | strong scaling | weak scaling |
|---|---|---|
| linear grid size $N_x$ | 16, 32, 64, 128 | 16, 20, 25, 32, 40 |
| CPU node count $P$ | 1, 2, 4, 8, 16 | 1, 2, 4, 8, 16 |
| GPU count $P$ | 1, 2, 4, 8 | 1, 2, 4, 8 |

they were approximated to the nearest integer. For example, at $P = 1$, $N_x = 16$, and $N = 4096$. Then, for $P = 2$, $N$ must be double the previous amount, $N = 8192$. Therefore, $N_x$ is chosen such that $N_x = \sqrt[3]{8192} = 20.158... \approx 20$, and so on.

Each data point in the figures of the results represents the mean of three independent SLURM jobs with the error bar being the standard deviation across runs. The job scripts can be found under `job-scripts/strong-scaling*.sh` and `job-scripts/weak-scaling*.sh`, whereas the results, the log files, and the plotting scripts for all the runs performed in this chapter can be found under `results/scaling-studies` of the Gitlab repository for this project.

Given a single-rank wall time

$$T_1$$

and the parallel time

$$T_P$$

, the speedup $S_P$ and efficiency $E_P$ are defined as,

$$S_P = \frac{T_1}{T_P}, \qquad E_P = \frac{S_P}{P}. \tag{4.1}$$

Ideal strong scaling corresponds to $S_P = P$ (or $E_P = 1$) (a linear graph), whereas weak scaling keeps $T_P$ constant (a horizontal line) [6].

### 4.3.2 Scaling Results CPU

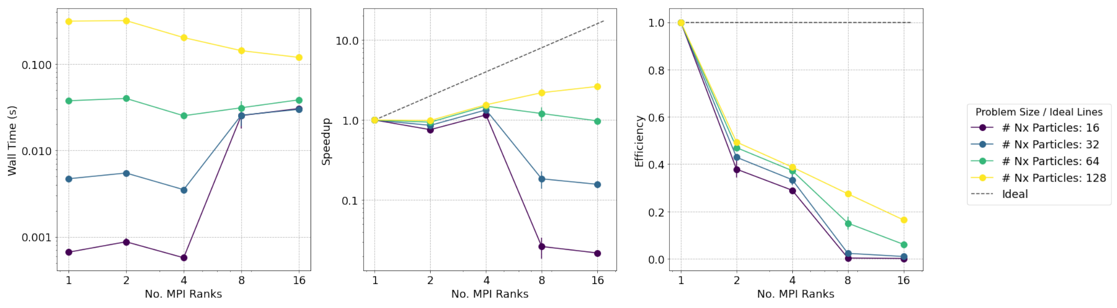

Figure 4.1: CPU strong scaling where the problem size remains constant with increasing MPI ranks. $N_x$ is the number of particles set along one axis, where the total number of particles in the simulation is $N = N_x^3$. From left to right, (a) Wall time as a function of MPI ranks, (b) Speedup as a functino of MPI ranks (c) Efficiency as a function of MPI ranks. Speedup and efficiency defined by Equation 4.3.1

Figure 4.1 shows that our strong-scaling experiment on CPUs performs initially better, but then worsens when increased from 4 to 8 ranks. This is true except for the case of $N = 128^3$, where the Speedup is far from the ideal case but still improves for each increase in ranks as seen by the yellow line in Figure 4.1(b). The efficiency plot in Figure 4.1(c) is also far from the ideal $E = 1$ line, where instead by the final runs for all problem sizes it is closer to 0.

The most likely culprit for this is the communication time for the MPI messages. There is evidence for this in Figure 4.1(a), which shows that wall time increases for small problem sizes, but by the largest problem size of 128 particles it is decreasing, although still not ideally. The time for an MPI message to be sent/received, $t_{\text{mpi}}$ can be expressed by,

$$t_{\text{mpi}} = \alpha + \beta L \tag{4.2}$$

where $\alpha$ represents the latency (start up time) of sending/receiving a message, and $\beta L$ represents the time to actually move the message buffer across ranks, where $L$ is the size of the buffer in bytes [6].

A full $128^3$ grid contains about $2.1 \times 10^6$ complex modes (roughly $1.0 \times 10^6$ Hermitian pairs), a problem size that can be considered relatively small when implemented on a high performance computing cluster such as merlin6. In c++, one complex value holds two 64-bit doubles and occupies 16 bytes, and a pair therefore occupies 32 bytes, so all pairs together occupy $\sim 32$ MB of raw data. On Merlin 6 the InfiniBand EDR link delivers 100 Gb s$^{-1}$ [7], at such rates a 32 MB buffer moves in roughly 2–3 ms. Once the grid is split across ranks the per-rank chunk is smaller still — for example, with $P = 16$ CPU ranks each rank handles $\frac{1}{16}$ of those pairs, about 2 MB, and 0.1 - 0.2ms per rank. At that point the start-up latency (which can be a few milliseconds) can outweigh the time needed to stream 2 MB, so the run becomes latency-bound. This numerical example describes the scenario for $128^3$ particles, yet in Figure 4.1 we have problem sizes of as low as $16^3$ particles which is where this problem is portrayed most clearly. When increasing the number of MPI ranks from 4 to 8, the wall time increases in more than one order of magnitude. Truly large initial condition generators in cosmology push $512^3$–$1024^3$ grids, where the bandwidth term $\beta L$ from Equation 4.3.2 should overtake latency. In such a case, we would expect that the wall time scales exclusively with problem size, but decreases with MPI ranks per problem size as opposed to what is present in the current results.

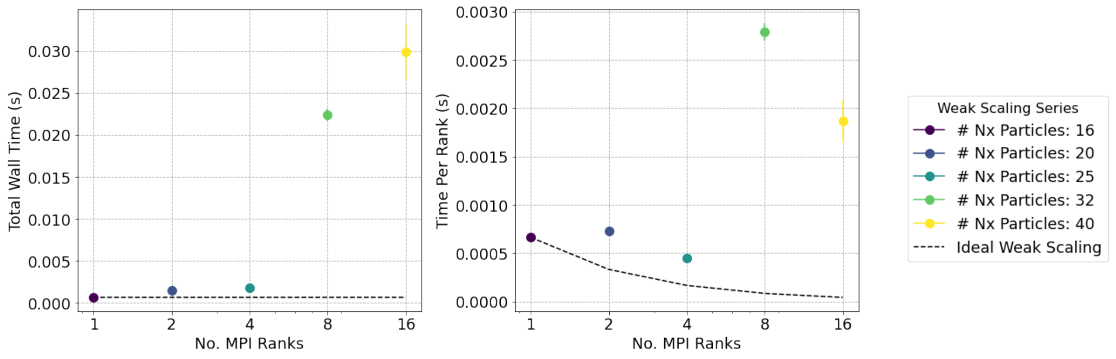

Figure 4.2: CPU weak scaling where the problem size increases with the number of MPI ranks. $N_x$ is the number of particles set along one axis, where the total number of particles in the simulation is $N = N_x^3$. (a) Left: total wall time, (b) Right: time per rank.

The weak scaling results (Figure 4.2) follow the same narrative, and they show a strong deviation from the ideal lines. In weak scaling, total wall time should remain constant across CPUs as problem size increases, and the time per rank should decrease proportionally to $1/P$.

Although the local workload is held roughly constant at ≈4000 particles, the wall time creeps upward with each doubling of rank count, indicating that communication and synchronisation, as well as possible load imbalance introduced by non-exclusive scheduling, dominate the run time. The sharp jump in Figure 4.2 is exactly what you expect when the fixed-latency part of the communication cost suddenly becomes larger than the work each rank has to do.

### 4.3.3 Scaling Results GPU

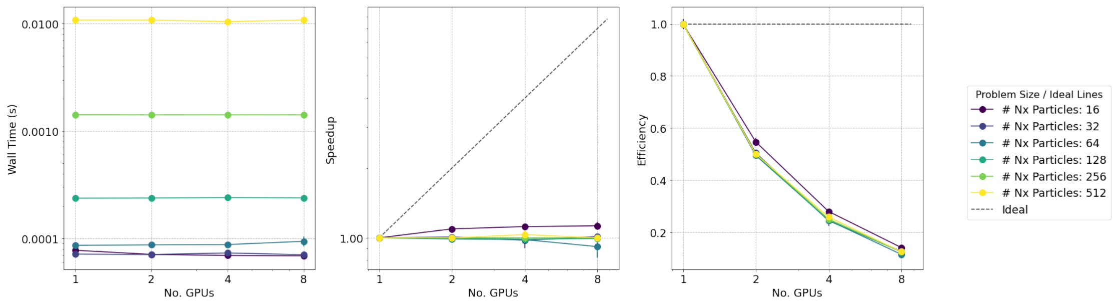

Figure 4.3: GPU strong scaling where the problem size remains constant with increasing GPUs (one MPI rank per GPU). $N_x$ is the number of particles set along one axis, where the total number of particles in the simulation is $N = N_x^3$. From left to right, (a) Wall time as a function of GPUs, (b) Speedup as a function of GPUs, (c) Efficiency as a function of GPUs. Speedup and efficiency defined by Equation 4.3.1

As seen in Figure 4.3(a), for every fixed problem size the wall time remains constant as we move from one to eight GPUs. The speed-up $S_P$ in Figure 4.3(b) remains around 1, and the efficiency $E_P$ in Figure 4.3(c) decreases with the device count increase. This is likely due to the lack of a large enough problem size to occupy any single GPU. We do not see the same issue as we did in the CPU scaling results because each run has one MPI rank per GPU, all kernels run entirely on device and the Hermiticity check therefore is performed on local memory, not requiring any cross rank communication. Because there are no calls to MPI send or receives, there are no latency or bandwidth problems to consider.

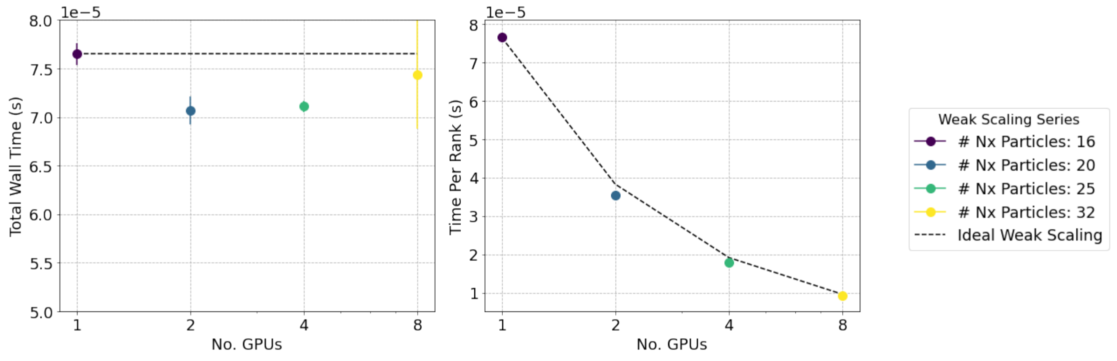

Figure 4.4: GPU weak scaling where the problem size increases with the number of MPI ranks. $N_x$ is the number of particles set along one axis, where the total number of particles in the simulation is $N = N_x^3$. (a) Left: total wall time, (b) Right: time per rank.

In the weak-scaling results seen in Figure 4.4 (a) the total wall time stays constant within a few tens of microseconds when the problem and the number of GPUs grow, and Figure 4.4 (b) shows the "time per rank" panel falls proportional to $1/P$. The drop is numerically small because the absolute times are already almost in the microsecond range, but the trend confirms that the GPU version should scale as expected once the per-device workload is large enough.

In summary, the GPU runs do not suffer the dramatic slow-downs seen on CPUs because each rank operates on its own device with no cross-rank messaging. At the same time, the problem sizes used here are too small to benefit from additional devices. While speed-up remains flat and efficiency drops. These results allow us to conclude that larger problem sizes should scale efficiently.

# Chapter 5

# Discussion

## 5.1 Physical Model and Limitations

The IPPL cosmo mini-app currently builds its displacement field with the first-order Zel'dovich approximation [3], just as the original NGenIC code that produced the Millennium and Millennium-XXL initial conditions [8, 9]. The next accuracy step is to add second-order Lagrangian perturbation theory (2LPT), which reduces transients and reproduces the target linear power spectrum more closely to the expected result. A public 2LPT extension of NGenIC exists [10], and as next steps the IPPL framework can implement the same correction terms.

## 5.2 Performance Portability

Unlike traditional initial condition generators built to run on CPU (N-GenIC [8], 2LPTic [10], MUSIC [11]) that write large snapshot files, our IC generator runs in-memory inside the IPPL application, using Kokkos for on-device execution (CPU or GPU), HeFFTe for distributed FFTs, and MPI for inter-rank communication. This design has the potential to remove the bottlenecks reported in recent simulations [12], such as the 200'000 CPU hours that were needed to read in the initial conditions in the Outer Rim simulation [13]. Importantly, it is already solving an observed bottleneck in the initial IPPL cosmo-app - when running the simulation on GPU the file read in time can account for up to 60% of the total run time [2]. Now the IC generator is already directly integrated with the simulation code, meaning there is zero read-in time to consider. Furthermore, this structure removes the complexity of having to compile and run different codes, replacing the workflow with just one input file in which you specify all the physics parameters of the simulation and then you can run it off the bat. This is inline with the approach used in the PKDGRAV3 simulation run on the Piz Daint supercomputers at CSCS in Switzerland, where the initial conditions are also generated in memory [14]. The current distributed layout is scalable to extreme particle counts and provides the performance portability required for next-generation cosmology simulations.

## 5.3 Hermiticity Check and Physical Validation

One unique feature of our implementation is the inclusion of an explicit Hermiticity check. Since the density field is constructed in Fourier space using complex amplitudes, but must yield a real-valued field in physical space, the following symmetry must hold:

$$\delta(\mathbf{k}) = \delta^*(-\mathbf{k}) \tag{5.1}$$

This symmetry is required for the inverse Fourier transform to yield a real field, and it is non-trivial to test it in distributed implementations where different MPI ranks may hold complementary subsets of the Fourier grid. We use MPI non-blocking communication to verify that the symmetry is preserved across the global domain, thus validating the physical correctness of the generated field.

While most existing IC generators rely on the structure of real-to-complex FFTs to implicitly preserve Hermitian symmetry, they do not expose an explicit check for it. We believe that including this test provides a valuable correctness guarantee, particularly in heterogeneous and GPU-targeted executions where floating-point and memory behaviour may differ.

Additional physical checks can be implemented on the generated initial conditions to ensure their validity. First, it is possible to recover the input power spectrum by computing $P_{\text{out}}(k)$ from the generated field and then to subtract it from the analytic or tabulated target $P_{\text{in}}(k)$. This value should ideally be equal to zero, but in practice smaller than some defined tolerance to ensure that the power spectrum has been preserved during initial condition generation.

Furthermore, the two-point correlation function, $\xi(r)$, is a statistical tool commonly used in cosmology that measures the probability of finding pairs of particles/galaxies separated by a distance $r$ compared to a random distribution [15]. The fourier transform of the two-point correlation function is the power spectrum. Therefore, in the IPPL initial condition generator $\xi(r)$ can be directly derived from the input power spectrum $P(k)$. For a white-noise spectrum ($P(k) = 1$), the expected $\xi(r)$ is a delta function such that the result is non-zero only at $r = 0$ and vanishing elsewhere. Adding this test would provide confirmation that no artifacts seeped into the generation of the gaussian random field (ie. it is truly random).

The final test to check the IPPL cosmo initial conditions generator is a comparison with the current state-of-the-art generators. In particular, given that the initial cosmo app for IPPL was built using NGENIC, the same power spectrum can be given to both the IPPL generator and NGENIC and the position and velocity distributions can be compared. Given that the results align, we can be confident that the IPPL cosmo initial conditions generator is following the expected physics behaviour. In the present time, the initial conditions generator of IPPL is still being built, so this comparison has not yet been made.

## 5.4 Scaling Discussion

The scaling results presented in Section 4.3 reveal significant deviations from ideal strong and weak scaling behaviour across CPU architectures. The GPU architectures also differ from ideal behaviour, but this was mainly due to not having the resources to test on a large enough problem size where the full benefit of the hardware is seen. In CPU strong scaling runs, the observed performance degradation, particularly the increase in wall time with increasing parallelism for many configurations, strongly indicates that the application suffers from substantial communication overheads that negate the benefits of additional processing elements.

# Chapter 6

# Conclusion

We have developed an initial condition generation method embedded within the IPPL framework that achieves full performance portability and parallel scalability. While the current implementation focuses on a $\Lambda$CDM universe with first-order Zel'dovich perturbations, its design permits future inclusion of higher-order physics.

Compared to widely-used tools like N-GenIC, MUSIC, and 2LPTic, our approach eliminates the file I/O bottleneck and leverages modern HPC libraries (Kokkos and HeFFTe) to support execution on both CPU and GPU nodes. This makes the method suitable for future exascale simulations requiring trillions of particles. Furthermore, the explicit Hermiticity check we implement adds a layer of physical validation absent in most current tools, and opens the door for additional tests on the generated fields. In its next phase, this initial conditions generator can be expanded to support 2LPT displacements and statistical validation against the input power spectrum.

In summary, state-of-the-art initial condition generation methods combine robust physics and efficient computation. Codes like N-GenIC, 2LPTic, and MUSIC set the standard for accuracy and recent trends focus on scaling up performance to run large scale cosmological simulations on GPU. Our implementation follows this frontier by being performance-portable and by eliminating file I/O bottlenecks, all while incorporating physical validation (a hermiticity check and power-spectrum confirmation) to ensure the generated fields are scientifically sound. This approach not only saves time and resources on current HPC systems, but also positions us well for future large scale cosmological simulations, where every bit of efficiency and accuracy will count.

# Appendix A

# Running Initial Conditions Simulation

## A.1 Compiling IPPL

### A.1.1 Environment modules

For this project, IPPL version 3.2.0 was compiled and run with the following modules on the PSI Cluster Merlin6

- for **CPU / serial build**: module load `cmake/3.25.2` `gcc/10.4.0` `cuda/11.5.1` `libfabric/1.18.0` `openmpi/4.1.4_slurm` `gsl/2.7` `hdf5/1.10.8_slurm`
- for **GPU build**: module load `gcc/12.3.0` `cuda/12.2.0` `openmpi/4.1.5_slurm` `hdf5/1.10.8_slurm` `libfabric/1.18.0` `cmake/3.25.2`

The module `libfabric` was included because of an update to the merlin6 cluster during the time of the project. In principle, this package does not need to be separately loaded for the code to run.

### A.1.2 CMake configuration (Merlin 6)

After loading the previous modules, IPPL was compiled using the following commands in the Merlin6 PSI cluster. More information on how to compile IPPL for your system can be found in the README.MD file of the IPPL library.

- **Serial (single rank)**

```
cmake .. -DCMAKE_BUILD_TYPE=Debug -DCMAKE_CXX_STANDARD=20 \
        -DIPPL_ENABLE_TESTS=True -DKokkos_VERSION=4.2.00 \
        -DIPPL_ENABLE_COSMOLOGY=True

make -j20
```

- **CPU / OpenMP**

```
cmake .. -DCMAKE_BUILD_TYPE=Debug -DCMAKE_CXX_STANDARD=20 \
        -DIPPL_ENABLE_FFT=ON -DIPPL_ENABLE_SOLVERS=ON \
        -DIPPL_ENABLE_COSMOLOGY=True -DIPPL_PLATFORMS=openmp \
        -DKokkos_VERSION=4.5.00

make -j20
```

- **GPU / CUDA**

```
cmake .. -DCMAKE_BUILD_TYPE=Release -DKokkos_ARCH_PASCAL61=ON \
        -DCMAKE_CXX_STANDARD=20 -DIPPL_ENABLE_FFT=ON \
        -DUSE_ALTERNATIVE_VARIANT=ON -DIPPL_ENABLE_COSMOLOGY=True \
        -DIPPL_PLATFORMS=cuda

make -j20
```

## A.2 Running the code

All job scripts can be found in the project's GitLab[1] under the `job-scripts` folder. If the directory is not visible in your GitLab view, kindly request access from *Andreas Adelmann* or *Sonali Mayani*.

For running the code, the following prerequisites should be met:

- `infile.dat` exists in the working directory of the job scripts – this file contains the simulation parameters (copies are under `job-scripts/input-file-examples` in the gitlab).
- `tf.dat` – transfer-function table - can be left empty for testing of the code (as was done in this project).

All of the provided jobscripts should run out of the box given that the code has been compiled successfully, and the directories in the job scripts are changed to match your own user directories.

### A.2.1 In serial

To run the code in serial you can call the following command

```
srun "$EXEC_DIR/StructureFormation" infile.dat tf.dat
out $DATA_DIR FFT 0.01 LeapFrog --overallocate 1.0 --info 5
```

replacing `$EXEC_DIR` with the directory leading to the executable files (eg. `../ippl/build-cosmo/cosmology-new`) and `$DATA_DIR` leading to the the directory to which the timings and data files should be saved (eg. `/data/user/user_name/` on merlin6).

### A.2.2 On CPU (MPI + OpenMP)

The batch script to run the code using openmp and/or mpi can be found in the gitlab under `job-scripts/job.sh`, and it is also provided below. The key changes that need to be made in the batch script to run for your own system are:

[1] `https://gitlab.ethz.ch/ebobrova/semester-project`

- `#SBATCH --nodes=5 --ntasks-per-node=1` ⇒ five MPI ranks, one per node. Change for your preferred simulation settings.
- `export OMP_NUM_THREADS=1` can be increased to also use multi-threading cores within a node (in which case `--hint=nomultithread` should be removed).
- `EXEC_DIR=.../ippl/build-cosmo-openmp/cosmology-new` should point to the CPU build and the directory to store the data in `$EXEC_DIR` should be set.
- Final `srun` passes the positional arguments `infile.dat tf.dat out $DATA_DIR FFT 0.01 LeapFrog`.

Listing A.1: CPU job script (excerpt)

```bash
#!/bin/bash
#SBATCH --cluster=merlin6      # compute partition
#SBATCH --partition=hourly
#SBATCH --time=00:05:00
#SBATCH --nodes=5              # 5 MPI ranks
#SBATCH --ntasks-per-node=1
#SBATCH --cpus-per-task=1
export OMP_NUM_THREADS=1
EXEC_DIR="/.../ippl/build-cosmo-openmp/cosmology-new"
DATA_DIR="/data/.../zeldovich_ICs"

srun "$EXEC_DIR/StructureFormation" \
     infile.dat tf.dat out $DATA_DIR FFT 0.01 LeapFrog \
     --overallocate 1.0 --info 5
```

### A.2.3 On GPU (MPI + CUDA)

The code was run on GPU on the PSI Gwendolen machine. Gwendolen has 1 node with 8 GPUs available. The batch script to run the code on GPUs can be found in the gitlab under `job-scripts/job-gpu.sh`, and it is also provided below. The key changes that need to be made in the batch script to run for your own system are:

- `#SBATCH --gpus=4 --ntasks=8` runs two MPI ranks per GPU. Adjust `--ntasks` to change the number of MPI ranks per GPU. The highest number of GPUs is 8 on Gwendolen.
- Build directory `$EXEC_DIR` (eg. `../ippl/build-cosmo-gpu/cosmology-new`) should point to the GPU build and the directory to store the data in `$EXEC_DIR` should be set.
- Final `srun` passes the positional arguments `infile.dat tf.dat out $DATA_DIR FFT 0.01 LeapFrog`.

Listing A.2: GPU job script (excerpt)

```bash
#!/bin/bash
#SBATCH --clusters=gmerlin6
#SBATCH --partition=gwendolen
#SBATCH --nodes=1
#SBATCH --gpus=4
#SBATCH --ntasks=8
```

```
#SBATCH --time=00:05:00
export OMP_NUM_THREADS=1
EXEC_DIR="../ippl/build-cosmo-gpu/cosmology-new"
DATA_DIR="/data/.../zeldovich_ICs"

srun "$EXEC_DIR/StructureFormation" \
     infile.dat tf.dat out $DATA_DIR FFT 0.01 LeapFrog \
     --overallocate 1.0 --info 5
```

## A.3 Running a test case

In order to run the test of the hermiticity function in `ippl/test/cosmology`, you must first reconfigure any of the above builds (from Appendix A.1) with the flag `-DIPPL_ENABLE_TESTS=ON` and rebuild before executing the test targets. Testing uses exactly the same workflow but links against the `build-*/test` target and executes `cosmology/TestHermiticity` instead of `StructureFormation`.

- **Serial run** `./TestHermiticity infile.dat tf.dat out ./DATA\_DIR FFT 1.0 LeapFrog`.
- **CPU multi-rank example** – job script `job-scripts/run_test.sh`
- **GPU example** – job script `job-scripts/run-gpu_test.sh`

Listing A.3: GPU test script (excerpt)

```
#!/bin/bash
#SBATCH --clusters=gmerlin6
#SBATCH --partition=gwendolen
#SBATCH --nodes=1
#SBATCH --gpus=4
#SBATCH --ntasks=1
#SBATCH --time=00:05:00
#SBATCH --output=test-gpu-%j.out      # Output log
#SBATCH --error=test-gpu-%j.err       # Error log

EXEC_DIR="/psi/.../build-cosmo-gpu/test"
DATA_DIR="/data/.../zeldovich_ICs"

srun "$EXEC_DIR/cosmology/TestHermiticity" \
     infile.dat tf.dat out $DATA_DIR FFT 1.0 LeapFrog \
     --overallocate 1.0 --info 5
```

Each test prints four truth values, two for deliberately Hermitian inputs and two for broken ones. The results will be printed in the output file in the current working directory.

## A.4 Scaling Studies

### A.4.1 On CPU

To perform the scaling studies on CPU, compile the code for CPU as described in Appendix A.1. Then copy the scaling scripts from the gitlab and replace the executive and data saving directories.

Listing A.4.1 shows how to use 16 CPUs on merlin6 for a strong scaling study where the problem size is kept constant and the number of CPUs is increased. Each node will use 1 cpu (therefore, 1 mpi rank per node). For each problem size ranging from $16^3$, to $128^3$ the code is executed on 1...16 nodes. You can easily add more problem sizes, or change the CPU configurations. Take care that the input directory contains the input file for the different specified problem sizes, named following the convention `infile{n}.dat`

Listing A.4: CPU Strong Scaling (excerpt)

```bash
#!/bin/bash
#SBATCH --cluster=merlin6
#SBATCH --partition=hourly
#SBATCH --time=00:15:00

#SBATCH --job-name=ic-strong-scaling
#SBATCH --output=ic-strong-scaling-%j.out # Output file
#SBATCH --error=ic-strong-scaling-%j.err  # Error file

#SBATCH --hint=nomultithread
#SBATCH --nodes=16
#SBATCH --ntasks-per-node=1
#SBATCH --cpus-per-task=1

export OMP_NUM_THREADS=1
export OMP_PROC_BIND=spread
export OMP_PLACES=cores

EXEC_DIR="../ippl/build-cosmo-openmp/cosmology-new"
INPUT_DIR="../scaling_studies"
DATA_DIR="..scaling_studies/results"

for n in 16 32 64 128; do
  for nodes in 1 2 4 8 16; do
    srun --nodes=$nodes "$EXEC_DIR/StructureFormation"
    "$INPUT_DIR/infile${n}.dat" "$INPUT_DIR/tf.dat" out
    "$DATA_DIR/results_strong/${n}_particles_" FFT 0.01 LeapFrog
    --overallocate 1.0 --info 5
  done
done
```

Weak scaling follows a similar structure but instead it increases the problem size with every increase in number of cpus, meaning only a single for loop is required. The problem size is determined by the total number of particles that there are, so every time the number of nodes is doubled, the total number of particles $N$ is doubled. However, in the code, this is represented by $n = N_x = \sqrt[3]{N}$, the number of particles along one axis. The full script can also be found in the gitlab, and as always the EXEC, INPUT and DATA directories seen in Listing A.4.1 should be set appropriately, and the user must ensure that all the correct input files exist in the input directory.

Listing A.5: CPU Weak Scaling (excerpt)

```bash
#!/bin/bash
#SBATCH --cluster=merlin6
#SBATCH --partition=hourly
#SBATCH --time=00:05:00

#SBATCH --job-name=ic-weak-scaling
#SBATCH --output=ic-weak-scaling-%j.out # Output file
#SBATCH --error=ic-weak-scaling-%j.err  # Error file

#SBATCH --hint=nomultithread
#SBATCH --nodes=16
#SBATCH --ntasks-per-node=1
#SBATCH --cpus-per-task=1

export OMP_NUM_THREADS=1

EXEC_DIR="/psi/home/bobrov_e/ippl/build-cosmo-openmp/cosmology-new"
INPUT_DIR="/psi/home/bobrov_e/cosmo-sim/runCosmo/scaling_studies"
DATA_DIR="/psi/home/bobrov_e/cosmo-sim/runCosmo/scaling_studies/results"

for nodes in 1 2 4 8 16; do
  case $nodes in
      1) n=16 ;;
      2) n=20 ;;
      4) n=25 ;;
      8) n=32 ;;
     16) n=40 ;;
  esac

  srun --nodes=$nodes "$EXEC_DIR/StructureFormation"
  "$INPUT_DIR/infile${n}.dat" "$INPUT_DIR/tf.dat"
  out "$DATA_DIR/results_weak/${n}_particles_"
  FFT 0.01 LeapFrog --overallocate 1.0 --info 5
done
```

### A.4.2 On GPU

On GPU, the scaling scripts are the same as those used above, but this time changing the slurm arguments to match the Gwendolen machine, and adjusting the nodes counts as there are only 8 available GPUs. This can be seen in Listings A.4.2 and A.4.2.

Listing A.6: GPU Strong Scaling (excerpt)

```bash
#!/bin/bash
#SBATCH --time=00:05:00
#SBATCH --nodes=1                         # One node
#SBATCH --ntasks=1                        # One MPI rank per GPU
#SBATCH --clusters=gmerlin6
#SBATCH --partition=gwendolen
#SBATCH --account=gwendolen
#SBATCH --gpus=8

#SBATCH --output=ic-strong-scaling-gpu-%j.out       # Output log
#SBATCH --error=ic-strong-scaling-gpu-%j.err        # Error log
#SBATCH --exclusive

export OMP_NUM_THREADS=1

EXEC_DIR="../ippl/build-cosmo-gpu/cosmology-new"
INPUT_DIR="../scaling_studies"
DATA_DIR="../scaling_studies/results/results_strong_gpu"

for n in 16 32 64 128 256 512; do
  for gpu in 1 2 4 8; do
    srun --gpus=$gpu "$EXEC_DIR/StructureFormation"
    "$INPUT_DIR/infile${n}.dat" "$INPUT_DIR/tf.dat" \
      out "$DATA_DIR/${n}_particles_${gpu}_" FFT 0.01
      LeapFrog --overallocate 1.0 --info 5
      --kokkos-map-device-id-by=mpi_rank
  done
done
```

Listing A.7: GPU Weak Scaling (excerpt)

```bash
#!/bin/bash
#SBATCH --time=00:05:00
#SBATCH --nodes=1                         # One node
#SBATCH --ntasks=1                        # One MPI rank per GPU
#SBATCH --clusters=gmerlin6
#SBATCH --partition=gwendolen
#SBATCH --account=gwendolen
#SBATCH --gpus=8

#SBATCH --job-name=ic-weak-caling-gpu
#SBATCH --output=ic-weak-scaling-gpu-%j.out       # Output log
#SBATCH --error=ic-weak-scaling-gpu-%j.err        # Error log
#SBATCH --exclusive

# Optional: set threading policies (can omit or adjust)
export OMP_NUM_THREADS=1

EXEC_DIR="../ippl/build-cosmo-gpu/cosmology-new"
INPUT_DIR="../scaling_studies"
DATA_DIR="../scaling_studies/results/results_weak_gpu"

for gpu in 1 2 4 8; do
  case $gpu in
      1) n=16 ;;
      2) n=20 ;;
      4) n=25 ;;
      8) n=32 ;;
  esac

  srun --gpus=$gpu "$EXEC_DIR/StructureFormation"
  "$INPUT_DIR/infile${n}.dat" "$INPUT_DIR/tf.dat"
  out "$DATA_DIR/${n}_particles_${gpu}_" FFT 0.01
  LeapFrog --overallocate 1.0 --info 5
done
```

### A.4.3 Plotting Script

The plotting scripts used to generate the scaling results in Section 4.3 can be found in the gitlab under `results/scaling-studies/results/scaling_plots.ipynb`. It is a jupyter notebook which iterates through the folders of the current directory to extract the timings from all the files into a pandas data frame, take the averages of all the three runs per study, and then plots and saves all the figures discussed in the results section. For easiest use of this script, one should ensure that they have the same saving directory structure as that shown in the gitlab.

# Appendix B

# Replicating Simulation

## B.1 Compiling

In order to run the IPPL mini-app, we will need to install and compile the IPPL framework as well as NGENIC for the generation of initial conditions. IPPL can be compiled based on the instructions in Appendix A.1. If you would like to use the original version of the cosmo-app from [2] the cosmology flag should be replaced with `-DENABLE_COSMOLOGY=ON`. In principle, this is not necessary as the read in method should also work in the updated code. The scripts below would have to be adapted to take in an input file from the command line.

Cloning and then following the `README.MD` file in the repository mentioned in [2], [1], will get you to generate the initial conditions from N-GenIC and then they can be plugged into the cosmology simulation by means of the `job-scripts/replication/structure-formation-32*.sh` scripts in the gitlab of this project (which are a copy from [2]). Two helper scripts have been created to aid in the generation of the initial conditions. The first is `job-scripts/replication/genic-job.sh` which is a job-script that you can run on a cluster to generate the initial conditions in parallel. The second is `merge_files.sh` which will help you merge all the IC files which were split up into $n$ ranks into a single data file.

[1] `https://github.com/bcrazzolara/SimGadget`

# Bibliography

[1] M. Frey, A. Vinciguerra, S. Muralikrishnan, S. Mayani, V. Montanaro, and A. Adelmann, "IPPL-framework/ippl: IPPL 3.1.0." `https://github.com/IPPL-framework/ippl`, 2025. Computer software, version 3.2.0.

[2] B. Crazzolara, "Cosmological structure formation with the performance-portable ippl library," 2024. Accessed 12th May 2025.

[3] Y. B. Zel'dovich, "Gravitational instability: An approximate theory for large density perturbations," *Astronomy and Astrophysics*, vol. 5, pp. 84–89, Mar. 1970.

[4] R. Scoccimarro, "Transients from initial conditions: A perturbative analysis," *Monthly Notices of the Royal Astronomical Society*, vol. 299, no. 4, pp. 1097–1118, 1998.

[5] M. Crocce, S. Pueblas, and R. Scoccimarro, "Transients from initial conditions in cosmological simulations," *Monthly Notices of the Royal Astronomical Society*, vol. 373, no. 2, pp. 369–381, 2006.

[6] G. Hager and G. Wellein, "Efficient mpi programming," in *Introduction to High Performance Computing for Scientists and Engineers*, ch. 10, pp. 235–253, Boca Raton, FL: CRC Press, Taylor & Francis Group, 2011.

[7] "The merlin hpc cluster." `https://www.psi.ch/en/awi/the-merlin-hpc-cluster`, 2025. Accessed 9 July 2025.

[8] V. Springel, "NGenIC: Cosmological initial conditions generator." `https://www.h-its.org/2014/11/05/ngenic-code/`, 2014. Accessed 10 July 2025.

[9] R. E. Angulo, V. Springel, S. D. M. White, A. Jenkins, C. M. Baugh, and C. S. Frenk, "Scaling relations for galaxy clusters in the millennium-xxl simulation," *Monthly Notices of the Royal Astronomical Society*, vol. 426, pp. 2046–2062, 2012.

[10] S. Pueblas, "2LPTic: Second-order lagrangian initial condition generator." `https://cosmo.nyu.edu/roman/2LPT/`, n.d. Accessed 10 July 2025.

[11] O. Hahn and T. Abel, "MUSIC: Multi-scale cosmological initial-conditions." `https://www-n.oca.eu/ohahn/MUSIC/`, 2011.

[12] O. Hahn and T. Abel, "Multi-scale initial conditions for cosmological simulations," *Monthly Notices of the Royal Astronomical Society*, vol. 415, no. 3, pp. 2101–2121, 2011.

[13] K. Heitmann, H. Finkel, A. Pope, V. Morozov, N. Frontiere, S. Habib, E. Rangel, T. Uram, D. Korytov, H. Child, S. Flender, J. Insley, and S. Rizzi, "The outer rim simulation: A path to many-core supercomputers," *The Astrophysical Journal Supplement Series*, vol. 245, no. 1, p. 16, 2019.

[14] D. Potter, J. Stadel, and R. Teyssier, "Pkdgrav3: Beyond trillion particle cosmological simulations for the next era of galaxy surveys," *Computational Astrophysics and Cosmology*, vol. 4, no. 1, p. 2, 2017.

[15] "The two-point correlation function." NED/IPAC Lecture Notes, n.d. Provides an introduction defining $\xi(r)$ as the excess probability over random for galaxy clustering.