ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

PSI

Designing a Hermiticity Test for the Zel'dovich Linear Initialization in the 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 22, 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 the provided 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 using a hermiticity test function. A parallel hermiticity test was implemented to validate the physical consistency of density fields generated by the provided Zel'dovich initial conditions generator. Using a test power spectrum, the generated field was confirmed to satisfy the required Hermitian symmetry. The next step is to supply a physical power spectrum to the existing initialization framework and generate realistic initial conditions for use in cosmological simulations with the IPPL library.

Contents

1 Introduction **1**

2 Theoretical Background **3**

3 Methods **5**
3.1 Initial Conditions Generation 5
3.2 Hermiticity Check Function 7

4 Results **12**
4.1 Hermiticity Function Validation 12
4.1.1 Algorithmic description 12
4.1.2 Results 13
4.2 Scaling Studies 14
4.2.1 Scaling set-up 14
4.2.2 Scaling Results CPU 15
4.2.3 Scaling Results GPU 19

5 Discussion **22**
5.1 Physical Model and Limitations 22
5.2 Performance Portability 22
5.3 Hermiticity Check and Physical Validation 23
5.4 Scaling Discussion 23

6 Conclusion **25**

Appendix A Running Initial Conditions Simulation **26**
A.1 Compiling IPPL 26
A.1.1 Environment modules 26
A.1.2 CMake configuration (Merlin 6) 26
A.2 Running a Hermiticity Test 27
A.3 Scaling Studies 29
A.3.1 On CPU 29
A.3.2 On GPU 31
A.3.3 Plotting Script 32
A.4 Running the code 33
A.4.1 In serial 33
A.4.2 On CPU (MPI + OpenMP) 33
A.4.3 On GPU (MPI + CUDA) 34

Appendix B Replicating Simulation **35**
B.1 Compiling 35

List of Figures

4.1 CPU strong scaling wall time 16
4.2 CPU strong scaling speedup 16
4.3 CPU strong scaling efficiency 17
4.4 CPU weak scaling wall time 18
4.5 CPU weak scaling efficiency 18
4.6 GPU strong scaling wall time 19
4.7 GPU strong scaling speedup 19
4.8 GPU strong scaling efficiency 20
4.9 GPU weak scaling wall time 21
4.10 GPU weak scaling efficiency 21

List of Tables

4.1 Scaling studies parameters 14

List of Listings

3.1 Pseudocode for the parallel hermiticity check 8

4.1 TestHermiticity.cpp output 13
4.2 StructureFormation output snippet 13

A.1 CPU test script (excerpt) 28
A.2 GPU test script (excerpt) 28
A.3 CPU Strong Scaling (excerpt) 29
A.4 CPU Weak Scaling (excerpt) 30
A.5 GPU Strong Scaling (excerpt) 31
A.6 GPU Weak Scaling (excerpt) 32
A.7 CPU job script (excerpt) 33
A.8 GPU job script (excerpt) 34

Chapter 1

Introduction

The IPPL (Independent Parallel Particle Layer) library provides a performance-portable particle-mesh framework that runs 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 reading in of IC files produced by external tools and packages. In the implementation where IC are read in from an external file generated by N-GenIC [3], the read-in time accounts for more than 60% of the simulation when running the app for 512^3 particles on 8 GPUs [2]. By generating the initial conditions ourselves, the process can be parallelized through domain decomposition. In addition to this, 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 program 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 density field constructed in the newly developed IC generator that will be later employed by the IPPL-based cosmology mini-app. The first step of the IC generator is to make a Gaussian random field where the Fourier modes are dependent on the input power spectrum. The Gaussian random field produced in Fourier space must obey Hermitian symmetry, $\delta(k) = \delta^*(-\mathbf{k})$, so that its real-space counterpart is strictly real. Thereby, a function `isHermitian()` is created to check this physical property of the generated field. This hermiticity test is implemented in parallel.

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

The remainder of the thesis is organised as follows. Chapter 2 reviews the theoretical 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 parallel hermiticity check. 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. Chapter 5 discusses the physical assumptions, performance-portability aspects and observed scaling behaviour, and identifies space 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 a structure formation which assumes a universe that is made of a pressure-less, collision-less "dust" made entirely of cold dark-matter particles - the λCDM 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.

Given the Lagrangian coordinate of a particle position $\mathbf{q}$ at initial time t_0, the position at subsequent times can be written in terms of the Lagrangian displacement field $\mathbf{\Psi}(\mathbf{q})$,

$$\mathbf{x}(t) = \mathbf{q} + D_1(t)\,\mathbf{\Psi}(\mathbf{q}), \quad \mathbf{v}(t) = \dot{D}_1(t)\,\mathbf{\Psi}(\mathbf{q}), \tag{2.1}$$

with $D_1 \propto a(t)$ being the linear growth factor, where $a(t)$ is a dimensionless scale factor corresponding to the relative expansion or contraction of the universe over time [4]. In this approximation, structure formation comes from tiny Gaussian density perturbations seen during the inflation period. The initial particle displacements (represented as the displacement field) are obtained from a Gaussian random field consistent with the target matter power spectrum $P(k)$.

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 $\delta(\mathbf{k})$ with wave numbers $\mathbf{k} = (k_x, k_y, k_z)$ by

$$\mathbf{\Psi}(\mathbf{k}) = -i\,\frac{\mathbf{k}}{k^2}\,\delta(\mathbf{k}), \qquad \delta(\mathbf{k}) = \sqrt{P(k)}\,(G_1 + iG_2), \tag{2.2}$$

where G_1 and G_2 are independent Gaussian random variables with mean 0 and variance 1 and $P(k)$ is the power spectrum that describes the initial distribution of matter and energy in the universe. Here $k = \sqrt{k_x^2 + k_y^2 + k_z^2}$ is the magnitude of the wavenumber vector. The Fourier space density field is Hermitian, $\delta(-\mathbf{k}) = \delta^*(\mathbf{k})$, which guarantees real values for the position and velocity after obtaining $\Psi(\mathbf{q})$ through an inverse Fast Fourier Transform (FFT).

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 [5], as well as the significant improvements obtained with second–order Lagrangian perturbation theory (2LPT) [6]. 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 N-GenIC [3].

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 $\delta(\mathbf{k})$ 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

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

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:

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

Following Equation 2.2. 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.

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 value of $\delta(\mathbf{k})$. This tolerance is taken as the double-precision value as the field $\delta(\mathbf{k})$ is initiated as a complex double in the code. 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:

$$\Psi_d(\mathbf{k}) = i\frac{k_d}{k^2}\delta(\mathbf{k}), \quad d = (x, y, z) \tag{3.10}$$

An inverse FFT is applied to obtain $\Psi_d(\mathbf{q})$ on the grid. For each spatial component $d \in \{x, y, z\}$, $\delta(\mathbf{k})$ is multiplied by ik_d/k^2 (skipping the $k = 0$ mode), an inverse FFT is applied, and the real part of the result is written into `cfield_m`. A final loop perturbs the positions of the particles from the cell centers $\mathbf{q}$ to $\mathbf{x} = \mathbf{q} + \mathbf{\Psi}(\mathbf{q})$ and stores the corresponding velocities as by Equation 2.1.

This approach ensures that the code benefits from 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. At this moment in time the IC generation is under development using the approach described above, and this project is focused on checking the physicality of the first step - the density field initialization.

3.2 Hermiticity Check Function

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

Throughout the discussion, the global 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 via `tol = Kokkos::Experimental::epsilon_v<double>`, avoiding both floating precision errors that result in a difference greater than 0 and arbitrary user chosen thresholds.

The pseudo code in Listing 3.1 illustrates the approach of the hermiticity test function. 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. These four phases, along with the corresponding line numbers from Listing 3.1 are explained next.

(1) Local scan and send count (lines 1-37). After defining all the relevant variables, we loop over all local indices and compute the global indices. For every global (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 and an immediate hermiticity check is performed on the pair, changing the flag `localHermitianFlag` to 0 in case of a violation.
2. $-\mathbf{k}$ is owned by a different rank, and the array that counts how many items must be sent to each rank, denoted c, is atomically incremented at the relevant index r corresponding to the owner rank. This increment is done 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), the local flag is set to 0 and an error message is printed.

If `nranks == 1`, then a local reduction is performed on the `localHermitianFlag` and 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 parallel hermiticity check

```
DEFINE struct HermitianPkg = (i,j,k, field(i,j,k))

function isHermitian()

    SET myrank TO current MPI_Rank()
    SET nranks TO MPI_Size()
    SET (Nx,Ny,Nz) TO global grid sizes

    // store message counts per rank (Equation 3.12)
    SET sendCount[] TO empty array

    /* --- Pass 1: count remote pairs & test local ones --- */
    SET localFlag TO 1
    parallel reduce each local (i,j,k) do
        if (i,j,k) = (0,0,0) then continue //skip DC mode

        // determine negative index based on Equation 3.1
        SET kneg TO Nx - i, Ny - j, Nz - k
        SET owner TO rank that owns kneg

        SET tol TO Kokkos machine precision of double
        if owner = myrank then    // partner is local
            // directly check hermiticity
            if |field(kneg) - conj(field(i,j,k))| > tol then
                localFlag = 0
            end if
        else    // partner is remote
            // increment send count array
            atomic_fetch_add(sendCount[owner],1)
        end if
        REDUCE localFlag across all threads
    end for

    // if only 1 rank, no communication needed
    // directly return the hermiticity result
    if nranks == 1 then
       return (localFlag != 0)

    /* allocate device buffers using sendCount prefix sums */
    SET send_disp[ ] TO displacement array // (Equation 3.13)
    SET total_sends TO total number of messages to send // (Equation 3.11)

    // allocate send and receive buffers
    SET sendBuffer[total_sends], recvBuffer[total_sends] TO empty array

    /* --- Pass 2: pack messages to send --- */
    parallel for each local (i,j,k) do

        // determine negative index based on Equation 3.1
        SET kneg TO Nx - i, Ny - j, Nz - k
        SET owner TO rank that owns kneg

```

```
        if owner != myrank then
            SET slot TO send_disp[owner] +
                atomic_fetch_add(sendCount[owner],1) // (Equation 3.14)
            // add message to correct slot in buffer
            SET sendBuffer[slot] TO HermitianPkg(i,j,k)
        end if
    end for

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

    /* --- Pass 3: verify remote pairs --- */
    parallel reduce idx = 0 .. total_sends-1 do
        // unpack HermitianPkg from recvBuffer
        SET p TO recvBuffer[idx]
        SET kneg_local TO (Nx-p.kx, Ny-p.ky, Nk-p.kz) // Equation 3.1

        SET delta_k TO field(kneg_local)
        SET delta_ck TO conj(delta_k)
        SET delta_neg_k TO (p.re, p.im) // from HermitianPkg

        SET tol TO Kokkos machine precision of double
        // check hermitian condition
        if |delta_ck - delta_neg_k| > tol then
            localFlag = 0
        end if
        REDUCE localFlag across all threads
    end for

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

(2) Host prefix scan (lines 39-41). If there are multiple ranks, then hermitian pairs need to be communicated across ranks. The send count array c contains the number of messages to send c_r to each rank r, where r is the index of the array (ie. index 0 corresponds to rank 0 and so on). This count array 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, which gives the starting index s_r of rank r's data within the contiguous send buffer (again, r is the index of the array corresponding to the rank being sent to). This is achieved by performing a prefix sum over the count array c. The total number of items to be sent is then given by

$$n_{\text{send}} = s[n_{\text{ranks}} - 1] + c[n_{\text{ranks}} - 1] \tag{3.11}$$

This total defines the size of the flat buffer. The displacement array ensures that data from different ranks are written into nonoverlapping contiguous regions. To illustrate, consider an example with 5 ranks, where the count array for messages sent from the fifth rank (rank 4) is

$$c = [c_0, c_1, c_2, c_3] = [3,\ 2,\ 4,\ 1] \tag{3.12}$$

Computing the prefix sum gives the prefix sum array:

$$s = [s_0, s_1, s_2, s_3] = [0,\ 3,\ 5,\ 9] \tag{3.13}$$

The flat send buffer will then be indexed as follows:

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

Here, each label `R0`, `R1`, `R2`, `R3` represents a message destined for rank 0, 1, 2, and 3, respectively. Messages to the same rank are stored contiguously, allowing for efficient communication using per-rank displacements. 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 occurs on the 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.

(3) Packing on the device (lines 43-59). Once the total size n_{send} and displacements s 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 $\delta(\mathbf{k})$ 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. Thereby we have

$$\text{slot} = s[r] + c[r], \tag{3.14}$$

where s_r is the start index (displacement) of messages destined for rank r, and c_r is a per-destination rank 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 61-84). 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_Irecv` 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 [7]. 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.

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 ε.

Summary The `isHermitian()` function:

1. Iterates over the local indices and identifies the global negative partner for each mode.
2. In the multi-rank case, it creates a count array and packs a send buffer with all relevant messages (indices and field values). These values are then exchanged using non-blocking MPI communication. In the single-rank case, this step is skipped.
3. Compares each complex pair up to the numerical tolerance ε.
4. Returns a boolean value indicating whether Hermitian symmetry is satisfied.

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 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 `StructureFormationManager` 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 from the `StructureFormationManager` class that are bound by `ifdef` ensuring they are only accessible if the code is compiled with the test flag on. This way the private variables in the StructureFormationManager class remain protected in the case that a normal simulation is being run. Appendix A.1 and A.2 outline 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 $0.5 + 0.0\text{i}$ 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

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

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 $0.5 + 0.0\text{i}$ but its partner is deliberately assigned $-1.0 + 0.0\text{i}$, 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 $\sqrt{P(k)/2}, (G_{\text{re}} + iG_{\text{im}})$, and its partner is implicitly the complex conjugate, in essence 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.2.

In the cosmology code, the function `LinearZeldoInitMP()` will generate an initial condition (IC) density field as a complex Gaussian random field with a target power spectrum $P(k)$. When running a full cosmology simulation, if the generated 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.
```

In the Gitlab repository one can find scripts to run a cosmology simulation 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.1 and A.4 respectively. These scripts

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

can be used to later run a full cosmology simulation once the IC generator is complete with the possibilitiy to define or read in a physical power spectrum $P(k)$.

4.2 Scaling Studies

4.2.1 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 OpenMPI 4.1 was used for CPU. To see the full specifications of the compiled IPPL in this section see Appendix A.1, and to Appendix A.3 to see how to run the scaling studies.

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.

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
CPU 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 linear grid size N_x	16, 32, 64, 128, 256	16, 20, 25, 32
GPU count P	1, 2, 4, 8	1, 2, 4, 8

The strong scaling job scripts use a nested for loop where each problem size is run on every rank in the CPU/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 $N_x = 128$, and as IPPL uses one particle per cell, $N = 128^3 = 2,097,152$ particles in total. Larger problem sizes $N_x \geq 256$ resulted in an out of bounds memory error on the Merlin cluster setups. The weak scaling parameters were chosen in such a way that the total number of particles N_x^3 doubles with each increase in the node / GPU count. Given that the input file only allows for the choice for the amount of particles along one axis, e.g. N_x and this value is cubed to get the total number of

particles $N = N_x^3$ on the whole grid (with one particle per cell), these numbers are not whole numbers, so 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 linear speedup, with $S_P = P$ or $E_P = 1$ [7].

For weak scaling T_P should remain constant (a horizontal line in a plot of wall time vs processes) and the efficiency,

$$E = \frac{T_1}{T_P} \tag{4.2}$$

should remain at $E = 1$ [7].

4.2.2 Scaling Results CPU

Figure 4.1 shows that the total wall time for the hermiticity test in the strong-scaling experiment on CPUs first increases between 1-2 ranks indicating communication overhead. The wall time then decreases consistently for all problem sizes (i.e. the system benefits from the parallelization) from 2-4 ranks, and then worsens significantly once more when increased from 4 to 8 ranks. This is true except for the case of $N = 128^3$, where the speedup improves for each increase in rank, although still not following the ideal case, as seen by the yellow line in Figure 4.2. The efficiency plot in Figure 4.3 is also far from the ideal $E = 1$ line, where by the final runs for all problem sizes it is closer to 0. For the largest problem size with $N_x = 128$ the speedup continues to increase with more ranks, meaning the total runtime decreases, but the efficiency drops below 0.2. This indicates that each additional processor contributes progressively less to the overall performance, and the speedup is achieved at a high cost in parallel resources.

The most likely culprit for these is the communication time for the MPI messages. There is evidence for this in Figure 4.1, 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_{mpi} can be expressed by,

$$t_{\text{mpi}} = \alpha + \beta L \tag{4.3}$$

where α represents the latency (start up time) of sending/receiving a message, and βL represents the time to actually move the message buffer across ranks, where L is the size of the buffer in bytes [7].

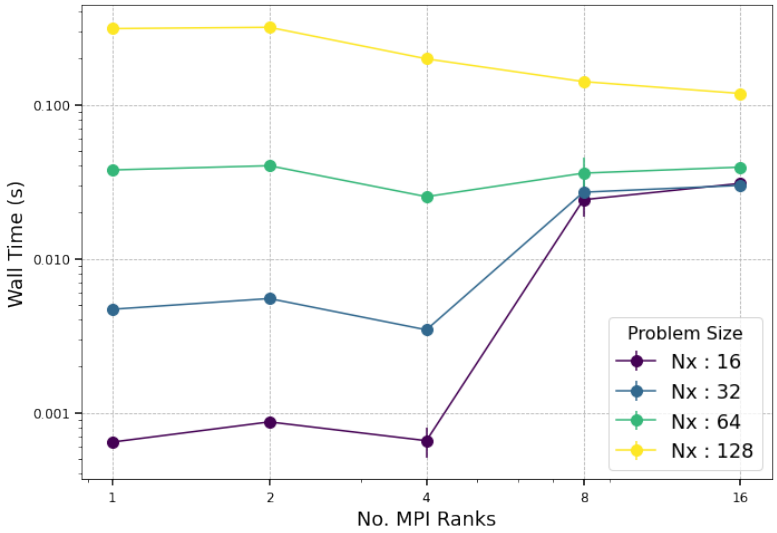

Figure 4.1: CPU strong scaling wall time as a function of MPI ranks. N_x is the number of particles set along one axis, with total particle count $N = N_x^3$.

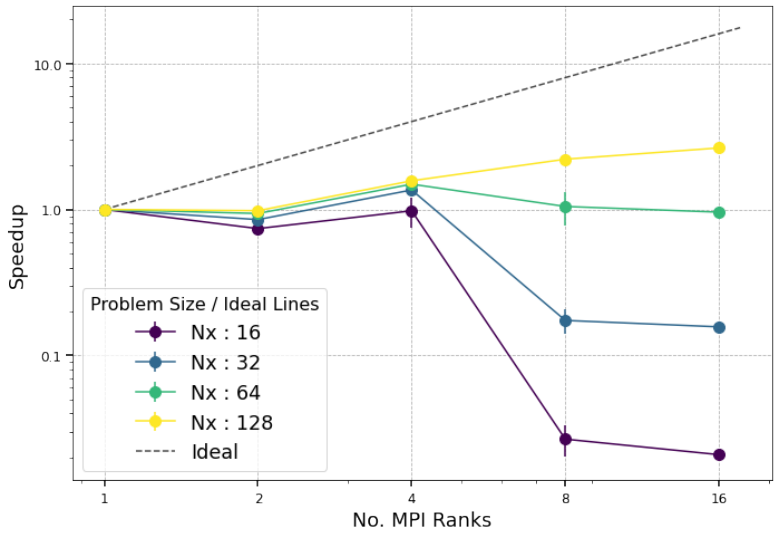

Figure 4.2: CPU strong scaling speedup as a function of MPI ranks. Speedup is defined by Equation 4.2.1. N_x is the number of particles set along one axis, with total particle count $N = N_x^3$.

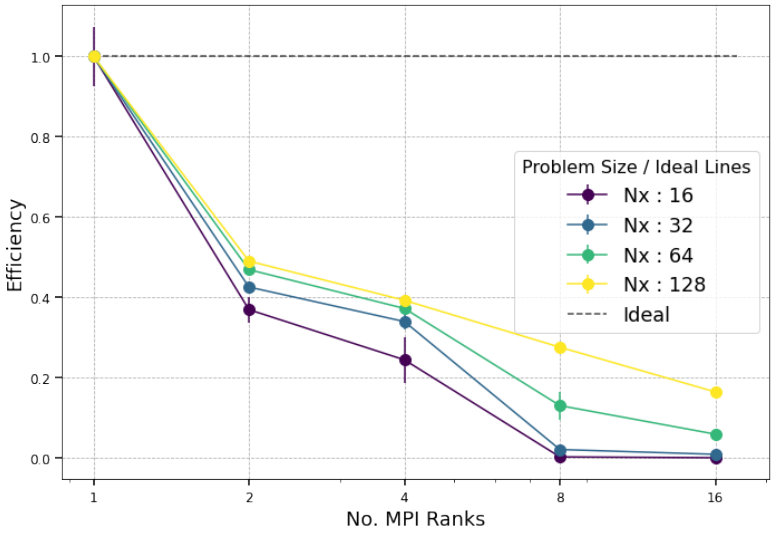

Figure 4.3: CPU strong scaling efficiency as a function of MPI ranks. Efficiency is defined by Equation 4.2.1. N_x is the number of particles set along one axis, with total particle count $N = N_x^3$.

A full 128^3 grid contains about 2.1×10^6 complex modes (roughly 1.0×10^6 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 ~ 32 MB of raw data. On Merlin 6 the InfiniBand EDR link delivers 100 Gb s^{-1} [8], 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 $P = 16$ each rank handles $\frac{1}{16}$ of those pairs, about 2 MB, taking 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 βL from Equation 4.2.2 should overtake latency. In such a case, we would expect that the wall time scales exclusively with problem size.

The weak scaling results (Figures 4.4 and 4.5) 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. Although the local workload is held roughly constant at $\approx$4000 particles, the wall time creeps upward with each doubling of rank count, indicating that communication and synchronisation dominate the run time. The sharp jump in Figure 4.4 is what you expect when the fixed-latency part of the communication cost suddenly becomes larger than the work each rank has to do, and the lack of stability in the efficiency plot (Figure 4.5) also supports this.

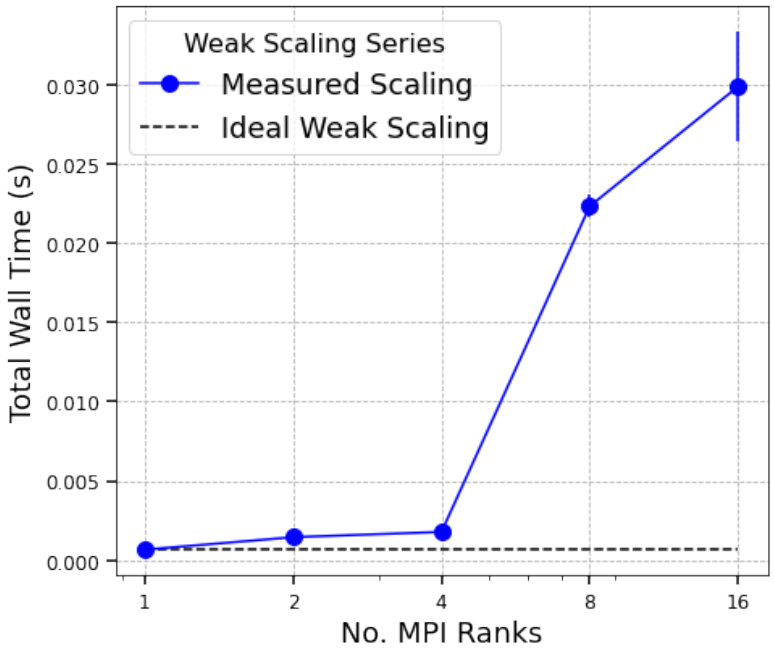

Figure 4.4: CPU weak scaling: total wall time as a function of MPI ranks.

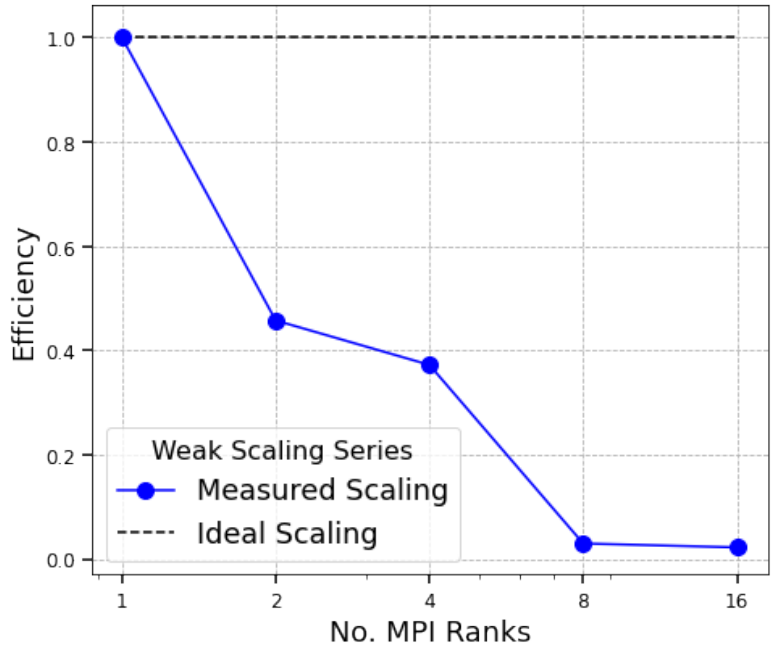

Figure 4.5: CPU weak scaling: efficiency as a function of MPI ranks. Efficiency is defined by Equation 4.2.1.

4.2.3 Scaling Results GPU

As seen in Figure 4.6, for every fixed problem size the total wall time on GPUs increases steadily as we scale from one to eight devices, with no performance improvement observed. The speedup, shown in Figure 4.7, consistently decreases across all problem sizes, and the efficiency in Figure 4.8 drops to nearly zero after the first GPUs are added.

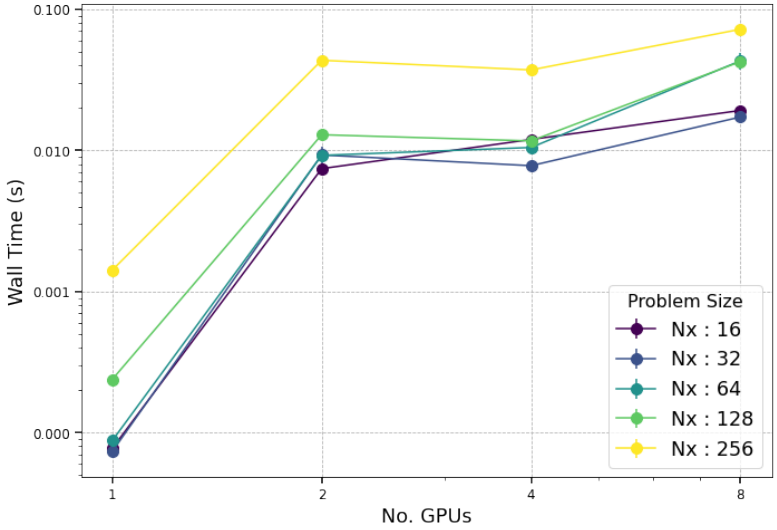

Figure 4.6: GPU strong scaling: wall time as a function of GPUs (one MPI rank per GPU). N_x is the number of particles set along one axis, with total particle count $N = N_x^3$.

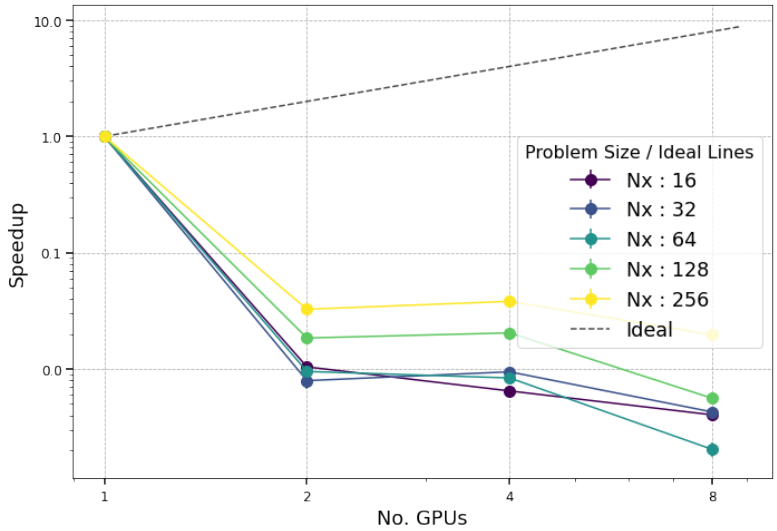

Figure 4.7: GPU strong scaling: speedup as a function of GPUs. Speedup is defined by Equation 4.2.1. N_x is the number of particles set along one axis, with total particle count $N = N_x^3$.

This behavior reflects poor scalability, and is even more severe than the CPU case. While in the CPU runs some limited gains were seen before overheads dominated, here the overhead dominates immediately. This is expected given the trends observed in the CPU strong scaling results and is consistent with findings in [2], which show that problem sizes on the order of $N = 256^3$ are barely sufficient to view an improvement in increasing GPUs. It is suspected that the problem size in the current runs is too small to fully utilize the available GPU parallelism. Furthermore, the dominant cost at higher GPU counts likely comes from communication overhead, since the required data exchange across ranks during setup introduces significant overhead, which outweighs any potential computational gains from parallelization at these problem sizes.

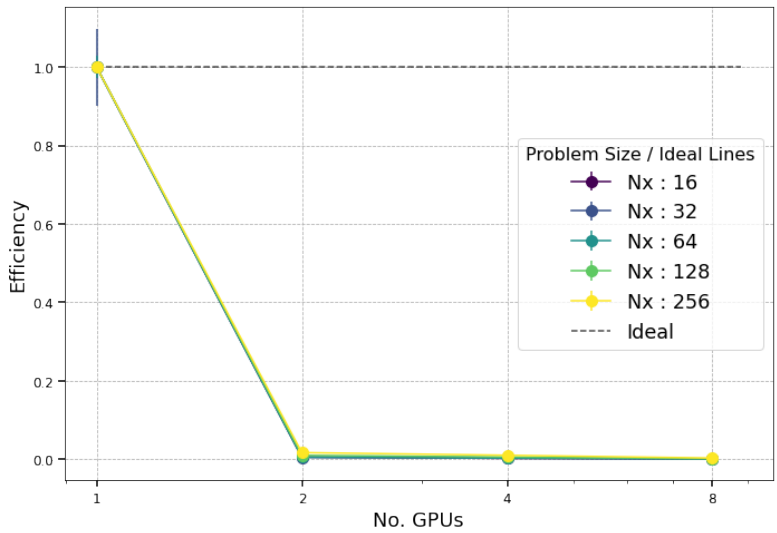

Figure 4.8: GPU strong scaling: efficiency as a function of GPUs. Efficiency is defined by Equation 4.2.1. N_x is the number of particles set along one axis, with total particle count $N = N_x^3$.

In the weak-scaling results shown in Figure 4.9, the total wall time increases by orders of magnitude as the number of GPUs grows, rather than remaining constant as expected in ideal weak scaling. Correspondingly, Figure 4.10 shows that the efficiency in weak scaling on GPUs, measured as $\frac{T_1}{T_p}$ drops sharply and almost immediately to zero, even for the smallest increases in GPU count. This poor scaling behavior mirrors the strong-scaling trends discussed earlier, and again points to the problem size being too small to effectively utilize GPU parallelism.

The GPU runs exhibit poor scaling behavior in both strong and weak scaling regimes. In the strong-scaling results, wall time increases with additional GPUs, speedup decreases, and efficiency drops to nearly zero after just a few ranks—indicating that the problem sizes are too small to benefit from GPU parallelism. Similarly, in the weak-scaling results, wall time increases by orders of magnitude and efficiency collapses almost immediately, rather than remaining constant as expected under ideal scaling.

Overall the hermiticity test is a communication heavy procedure with almost no computation being done. This makes it unlikley that the code can be improved to scale better since an increase in ranks will always result in an increase in communication. However, this segment of the code only needs to be run once at the beginning of the IC generation and still only accounts for a small fraction of total simulation time, especially when compared with particle read in time for codes without on the fly IC generation, as discussed in Section 5.2.

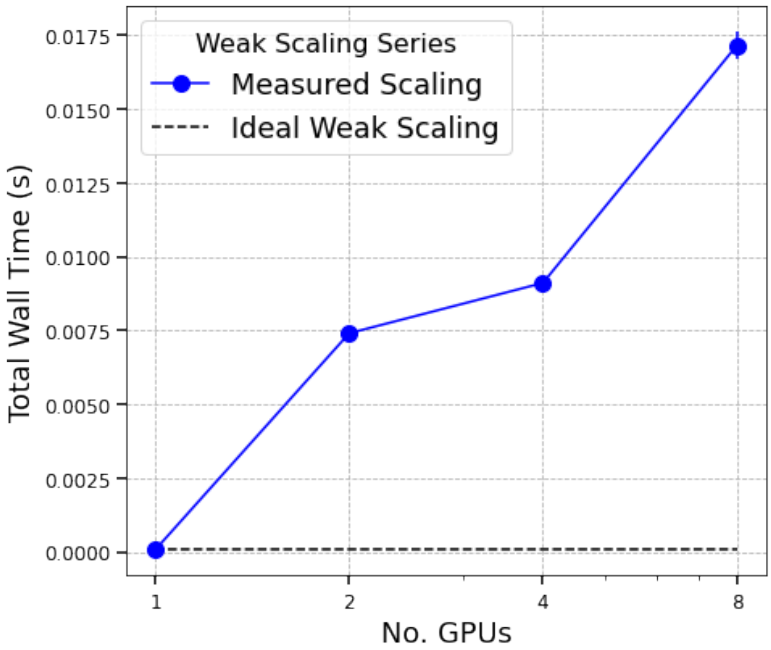

Figure 4.9: GPU weak scaling: total wall time as a function of MPI ranks (one rank per GPU).

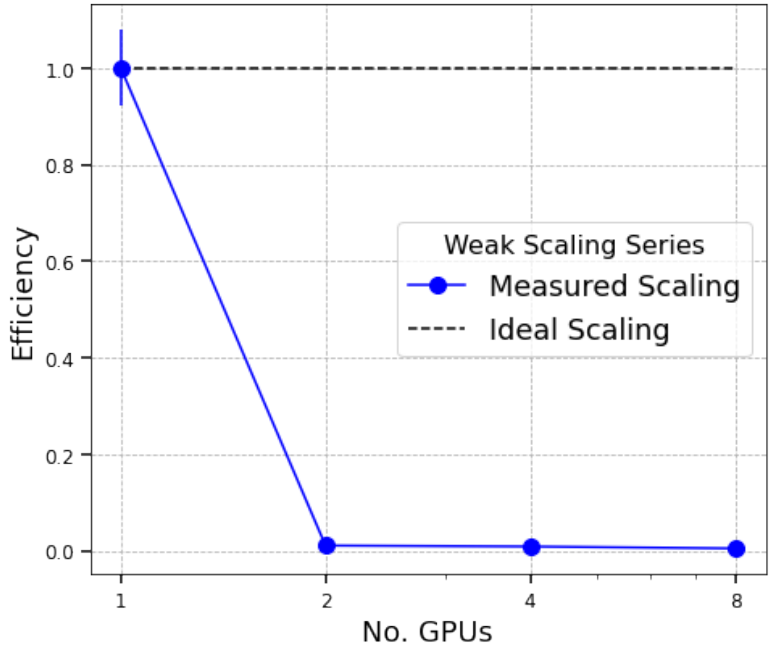

Figure 4.10: GPU weak scaling: efficiency (time per rank) as a function of MPI ranks.

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 [4], just as the original N-GenIC code that produced the Millennium and Millennium-XXL initial conditions [3, 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 N-GenIC 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 [3], 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]. When the IC generator will be directly integrated with the simulation code there will be no read-in time to consider. Although we still have to generate the initial particle positions and velocities, current test runs indicate that the timing will be comparable to other processes rather than dominate the simulation. 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 the pairs $\mathbf{k}$ and $-\mathbf{k}$ may lie on different MPI ranks holding 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 that needs to still be implemented to validate 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 N-GenIC, the same power spectrum can be given to both the IPPL generator and N-GenIC and the position and velocity distributions can be compared. In the scenario that the results align, we can be confident that the IPPL cosmo initial conditions generator will be 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 4.2 reveal significant deviations from ideal strong and weak scaling behaviour across CPU and GPU architectures. In CPU and GPU 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. However, in the full cosmological simulation, the hermiticity test needs only to run once when generating the ICs and makes up only a small fraction of the total run time of the simulation. For example, for a problem size of $N_x = 256$, the read in time from the IPPL Cosmology mini app on 8 GPUs of the file generated by N-GenIC is two orders of magnitude larger than the

hermiticity test for the same problem size [2]. Thereby, while achieving ideal scaling behaviour for this communication bound hermiticity test may be difficult, it is not believed that this will be a bottleneck for the timing of the full scale simulation.

Chapter 6

Conclusion

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

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 via a hermiticity check to ensure the generated fields are scientifically sound. This approach not only saves time and resources on current HPC systems, but also positions the IPPL cosmo mini app 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 a Hermiticity Test

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*. All scripts can be run on a cluster with a batch system using `sbatch job.sh`, replacing the name of the job file where necessary.

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).

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

- **Serial run** `./TestHermiticity infile.dat tf.dat out ./DATA_DIR FFT 1.0 LeapFrog`.
- **CPU multi-rank example** – Listing A.2 shows the job script `job-scripts/run_test.sh`
- **GPU example** – Listing A.2 shows the job script `job-scripts/run-gpu_test.sh`

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

Listing A.1: CPU test script (excerpt)

```
#!/bin/bash
#SBATCH --cluster=merlin6
#SBATCH --partition=hourly
#SBATCH --time=00:30:00

#SBATCH --job-name=test-multirank      # Job name
#SBATCH --output=test-multirank-30-%j.out # Output file
#SBATCH --error=test-multirank-30-%j.err  # Error file

#SBATCH --hint=nomultithread
#SBATCH --nodes=4
#SBATCH --ntasks-per-node=1
#SBATCH --cpus-per-task=1

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
```

Listing A.2: 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.3 Scaling Studies

A.3.1 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.3.1 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.3: CPU Strong Scaling (excerpt)

```
#!/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.3.1 should

be set appropriately, and the user must ensure that all the correct input files exist in the input directory.

Listing A.4: CPU Weak Scaling (excerpt)

```
#!/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.3.2 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.3.2 and A.3.2.

Listing A.5: GPU Strong Scaling (excerpt)

```
#!/bin/bash
#SBATCH --time=00:05:00
#SBATCH --nodes=1                          # One node
#SBATCH --ntasks=8                         # 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=$gpu --ntasks=$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.6: GPU Weak Scaling (excerpt)

```
#!/bin/bash
#SBATCH --time=00:05:00
#SBATCH --nodes=1                    # One node
#SBATCH --ntasks=8                   # 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=$gpu --ntasks=$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.3.3 Plotting Script

The plotting scripts used to generate the scaling results in Section 4.2 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.

A.4 Running the code

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. For this project, this workflow was tested for a static power spectrum with $P(k) = 1$ everywhere to make sure the hermiticity test is properly functional within the code base. This same workflow should work for running a full simulation once the IC generator is complete, and can be used to plot the positions and velocities generated to confirm that the initial distribution of particles matches the expected distribution.

A.4.1 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.4.2 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:

- `#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.7: CPU job script (excerpt)

```
#!/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" \
```

```
13      infile.dat tf.dat out $DATA_DIR FFT 0.01 LeapFrog \
14      --overallocate 1.0 --info 5
```

A.4.3 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.8: GPU job script (excerpt)

```
1  #!/bin/bash
2  #SBATCH --clusters=gmerlin6
3  #SBATCH --partition=gwendolen
4  #SBATCH --nodes=1
5  #SBATCH --gpus=4
6  #SBATCH --ntasks=8
7  #SBATCH --time=00:05:00
8  export OMP_NUM_THREADS=1
9  EXEC_DIR="../ippl/build-cosmo-gpu/cosmology-new"
10 DATA_DIR="/data/.../zeldovich_ICs"
11
12 srun "$EXEC_DIR/StructureFormation" \
13      infile.dat tf.dat out $DATA_DIR FFT 0.01 LeapFrog \
14      --overallocate 1.0 --info 5
```

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 N-GenIC 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." `https://amas.web.psi.ch/people/aadelmann/ETH-Accel-Lecture-1/projectscompleted/phys/blanca.pdf`, 2024. Accessed 21st Jul 2025.

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

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

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

[6] 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.

[7] 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.

[8] "The Merlin HPC Cluster." `https://www.psi.ch/en/awi/the-merlin-hpc-cluster`, 2025. Accessed 9 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/`. 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.

Acknowledgment

In this section I would like to thank Dr. Andreas Adelamann and my weekly supervisor Sonali Mayani. With the guidance of both Andreas and Sonali, I learned a lot, and quickly, about the intersection of physics and high-performance computing. I would also like to thank the time and effort that Sonali took to provide in-depth feedback on my report. Beyond my direct research project, I was warmly welcomed into the AMAS group at PSI, where I felt genuinely part of a collaborative scientific community. Overall, this was a formative experience as my first research project in Switzerland. I would also like to thank my 1.5 year old daughter Alexandra for sleeping (relatively) well these last 6 months.