Title

FlowPM: Distributed TensorFlow implementation of the FastPM cosmological N-body solver

Authors

Modi, C

Lanusse, F

Seljak, U

Publication Date

2021-10-01

DOI

10.1016/j.ascom.2021.100505

FlowPM: Distributed TensorFlow Implementation of the FastPM Cosmological N-body Solver

Chirag Modi[a], François Lanusse[b], Uroš Seljak[a,c]

[a]Berkeley Center for Cosmological Physics, Department of Physics, University of California, Berkeley, CA 94720, USA
[b]AIM, CEA, CNRS, Université Paris-Saclay, Université Paris Diderot, Sorbonne Paris Cité, F-91191 Gif-sur-Yvette, France
[c]Lawrence Berkeley National Laboratory, One Cyclotron Road, Berkeley, CA 94720, USA

Abstract

We present FlowPM, a Particle-Mesh (PM) cosmological N-body code implemented in Mesh-TensorFlow for GPU-accelerated, distributed, and differentiable simulations. We implement and validate the accuracy of a novel multi-grid scheme based on multiresolution pyramids to compute large-scale forces efficiently on distributed platforms. We explore the scaling of the simulation on large-scale supercomputers and compare it with corresponding Python based PM code, finding on an average 10x speed-up in terms of wallclock time. We also demonstrate how this novel tool can be used for efficiently solving large scale cosmological inference problems, in particular reconstruction of cosmological fields in a forward model Bayesian framework with hybrid PM and neural network forward model. We provide skeleton code for these examples and the entire code is publicly available at https://github.com/modichirag/flowpm.

Keywords:
PACS: cosmology: large-scale structure of universe, methods: N-body simulations

1. Introduction

N-Body simulations evolving billions of dark matter particles under gravitational forces have become essential tool for modern day cosmology data analysis and interpretation. They are required for accurate predictions of the large scale structure (LSS) of the Universe, probed by tracers such as galaxies, quasars, gas, weak gravitational lensing etc. Exact N-body simulations can provide very precise predictions and have been heavily optimized and tested over the last two decades (Teyssier, 2002; Nelson et al., 2015; Springel et al., 2020; Garrison et al., 2018). Despite this, they are extremely slow and computationally expensive, especially when one might require to run thousands of these simulations. As a result they are usually complemented by fast approximated numerical schemes such as Particle-Mesh (PM) solvers (Merz et al., 2005; Tassev et al., 2013; White et al., 2014; Feng et al., 2016; Izard et al., 2016; Monaco, 2016; Colavincenzo et al., 2018). In these approaches, the gravitational forces experienced by dark matter particles in the simulation volume are computed by Fast Fourier Transforms on a 3D grid. This makes PM simulations very computationally efficient and scalable, at the cost of approximating the interactions on small scales close to the resolution of the grid and ignoring the force calculation on sub-resolution scales.

The next generation of cosmological surveys such as the Dark Energy Spectroscopic Instrument (DESI) (DESI Collaboration et al., 2016), the Rubin Observatory Legacy Survey of Space and Time (LSST) (LSST Science Collaboration et al., 2009), and others will probe LSS with multiple tracers over unprecedented scales and dynamic range. This will make both the modeling and the analysis challenging even with current PM simulations. To improve the modeling in terms of dynamic range and speed, many recent works have explored applications of machine learning, with techniques based on deep convolutional networks and generative models (He et al., 2019; Kodi Ramanah et al., 2020). However these approaches often only learn the mapping between input features (generally initial density field) and final observables and completely replace the correct underlying physical dynamics of the evolution with end-to-end modeling. As a result, the generalization of these models across redshifts (time scales), length scales (cosmological volume, resolution etc.), cosmologies and observables of interest is limited by the training dataset and needs to be explored exhaustively to account for all failure modes.

At the same time, there has been a considerable interest in exploring novel techniques to push simulations to increasingly smaller and non-Gaussian regimes for the purpose of cosmological analyses. Recent work has demonstrated that one of the most promising approaches to do this is with forward modeling frameworks and using techniques like simulations based inference or reconstruction of cosmological fields for analysis (Seljak et al., 2017; Alsing et al., 2018; Cranmer et al., 2019). However given the highly non-linear forward dynamics and multi-million dimensionality of the cosmological simulations, such approaches are only tractable if one has access to differentiable forward simulations (Jasche and Wandelt, 2013; Wang et al., 2014; Seljak et al., 2017; Modi et al., 2018) wherein one can correctly analytically estimate the response (gradients) of cosmological probes such as final dark matter density field, dark

matter halos, galaxies etc. with respect to the input cosmology.

To address both of these challenges, in this work we present `FlowPM`, a TensorFlow implementation of an N-Body PM solver. Written entirely in TensorFlow, these simulations are GPU-accelerated and hence faster than CPU-based PM simulations, while also being entirely differentiable end-to-end. At the same time, they encode the exact underlying dynamics for gravitational evolution (within a PM scheme), while still leaving room for natural synthesis with machine learning components to develop hybrid simulations. These hybrids could use machine learning to model sub-grid and non-gravitational dynamics otherwise not included in a PM gravity solver.

As they evolve billions of particles in order to simulate today's cosmological surveys, N-Body simulations are quite memory intensive and necessarily require distributed computing. In `FlowPM`, we develop such distributed computation with a novel model parallelism framework - Mesh-TensorFlow (Mesh-TF) (Shazeer et al., 2018). This allows us to control distribution strategies and split tensors and computational graphs over multiple processors.

The primary bottleneck for large scale distributed PM simulations are distributed 3D-Fast Fourier Transforms (FFT), which incur large amounts of communications and make up for more than half the wall-clock time (Modi et al., 2019a). One way to overcome these issues is by using a multi-grid scheme for computing gravitational forces (Suisalu and Saar, 1995; Merz et al., 2005; Harnois-Déraps et al., 2013). In these schemes, large-scale forces are estimated on a coarse distributed grid and then stitched together with small scale force estimated locally. Following the same idea, we also propose a novel multi-grid scheme based on Multiresolution Pyramids (Burt and Adelson, 1983; Anderson et al., 1984), that does not require any fine-tuned stitching of scales, and benefits from highly optimized 3-D convolutions of TensorFlow.

In this first work, we have implemented the `FastPM` scheme of Feng et al. (2016) for PM evolution. Different PM schemes, as well as other approximate simulations like Lagrangian perturbation theory fields (Tassev et al., 2013; Kitaura et al., 2014; Stein et al., 2019), are built upon the same underlying components, with interpolations to-and-from PM grid and efficient FFTs to estimate gravitational forces being the key elements. Thus modifying `FlowPM` to include other PM schemes is a matter of implementation rather than methodology development. We choose to work with `FastPM` scheme for two reasons - i) it is the closest to a naive particle mesh simulation and avoids cosmology specific approximations such as decoupling linearly evolving large scales with perturbative models as done in schemes like COLA (Tassev et al., 2013). Therefore in this work, it allows us to focus in detail on the underlying components common to all PM schemes while only giving the outline of the full algorithm. ii) It has a differentiable python implementation that allows us to compare different experiments consistently. For further technical details on the choices and accuracy of the implemented `FastPM` scheme, we refer the reader to the original paper of Feng et al. (2016).

The structure of this paper is as follows - in Section 2 we outline the basic `FastPM` algorithm and discuss our multi-grid scheme for force estimation in Section 3. Then in Section 4, we introduce Mesh-TensorFlow. In Section 5, we build upon these and introduce `FlowPM`. We compare the scaling and accuracy of `FlowPM` with the Python `FastPM` code in Section 6. At last, in Section 7, we demonstrate the efficacy of `FlowPM` with a toy example by combining it with neural network forward models and reconstructing the initial conditions of the Universe. We end with discussion in Section 8.

Throughout the paper, we use "grid" to refer to the particle-mesh grid and "mesh" to refer to the computational mesh i.e. the geometry of connections between different processes and distribution of the tensors on them with Mesh-TF. However in section 6, we do refer the single grid force implementation as *mesh scheme*, with the use of *scheme* making the context clear. Unless explicitly specified, every FFT referred throughout the paper is a 3D FFT.

2. Particle Mesh N-body solver

A cosmological particle mesh simulation starts with sampling a Gaussian white-noise. This is scaled with the input linear power spectrum to generate the Gaussian initial conditions for the matter density field at high redshift. These are then used to generate the initial displacements and velocities of the particles at a very high redshift. This is usually done in PM codes (as in `FastPM` and `FlowPM`) with 2nd order Lagrangian perturbation theory (LPT). Schematically, a PM solver then implements a symplectic series of Kick-Drift-Kick operations (Quinn et al., 1997), following a leapfrog integration scheme, to do the time integration for gravitational evolution of the particles that leads to the observed large scale structures at the final time-step. In the *Kick* stage, we estimate the gravitational force on every particle and update their momentum (p). Next, in the staggered *Drift* stage, we displace the particle to update their position (x) with the current velocity estimate. The drift and kick operators can thus be defined as (Quinn et al., 1997) -

$$x(t_1) = x(t_0) + p(t_0)\mathcal{D}_{\rm PM} \tag{1}$$
$$p(t_1) = p(t_0) + f(t_0)\mathcal{K}_{\rm PM}, \tag{2}$$

where $\mathcal{D}_{\rm PM}$ and $\mathcal{K}_{\rm PM}$ are the scalar drift and kick factors and $f(t_0)$ is the gravitational force. For the leapfrog integration implemented in `FlowPM` (and `FastPM`), these operators are modified to include the position and velocity at staggered half time-steps instead of the initial time (t_0) in kick and drift steps respectively.

Estimating the gravitational force is the most computationally intensive part of the simulation. In a PM solver, the gravitational force is calculated via 3D Fourier transforms. First, the particles are interpolated to a spatially uniform grid with a kernel $W(r)$ to estimate the mass overdensity field at every point in space. The most common choice for the kernel, and the one we implement, is a linear window of unit size corresponding to the cloud-in-cell (CIC) interpolation scheme (Hockney and Eastwood, 1988). We then apply a Fourier transform to obtain the over-density field δ_k in Fourier space. This field is related

to the force field via a transfer function ($\nabla\nabla^{-2}$).

$$f(\mathbf{k}) = \nabla\nabla^{-2}\delta(\mathbf{k}) \tag{3}$$

Once the force field is estimated on the spatial grid, it is interpolated back to the position of every particle with the same kernel as was used to generate the density field in the first place.

There are various ways to write down the transfer function in a discrete Fourier space, as explored in detail in Hockney and Eastwood (1988). The simplest way of doing this is with a naive Green's function kernels ($\nabla^{-2} = k^{-2}$) and differentiation kernels ($\nabla = i\mathbf{k}$). In `FlowPM`, in addition to this kernel, we also implement a finite differentiation kernel as implemented in `FastPM`. This provides a sharper density fields and better cross-correlation with N-Body simulations for small number of time steps (see appendix of (Feng et al., 2016) for comparison)

$$\nabla^{-2} = \sum_{d=x,y,z} \left(x_0\omega_0 \mathrm{sinc}\frac{\omega_d}{2}\right)^2\Bigg)^{-1} \tag{4}$$

$$\nabla = D_1(\omega) = \frac{1}{6}\Big(8\sin\omega - \sin 2\omega\Big) \tag{5}$$

where x_0 is the grid size and $\omega = kx_0$ is the circular frequency that goes from $(-\pi, \pi]$.

In the current version, `FlowPM` implements this PM scheme with the `FastPM` kernels with force resolution of one i.e. the PM force is estimated on the same grid resolution as particle resolution. Unless otherwise specified, we have one particle per grid cell and hence the number of particles is the same as the grid size. At this force resolution, a 10 time-step `FastPM` scheme results in dark matter field which is 99% correlated with full N-Body (TreePM) simulations (of same particle resolution) up to k=1 h/Mpc and loose 5% power up to k=0.6 h/Mpc (Feng et al., 2016).

3. Multi-grid PM Scheme

Every step in PM evolution requires one forward 3D FFT to estimate the overdensity field in Fourier space, and three inverse FFTs to estimate the force component in each direction. The simplest way to implement 3D FFTs is as 3 individual 1D Fourier transforms in each direction (Pippig, 2013). However in a distributed implementation, where the tensor is distributed on different processes, this involves transpose operations and expensive all-to-all communications wherein each processor sends an individual message to every other processor in order to get the dimension being transformed on the same process. This makes these FFTs the most time-intensive step in the simulation (Modi et al., 2019a). There are several ways to make this force estimation more efficient, such as fast multipole methods (Greengard and Rokhlin, 1987) and multi-grid schemes (Merz et al., 2005; Harnois-Déraps et al., 2013). In FlowPM, we implement a novel version of the latter.

The basic idea of a multi-grid scheme is to use grids at different resolutions, with each of them estimating force on different scales which are then stitched together. Here we discuss this approach for a 2 level multi-grid scheme, since that is implemented in `FlowPM` and the extension to more levels is similar. The long range (large-scale) forces are computed on a global coarse grid that is distributed across processes. On the other hand, the small scale forces are computed on a fine, higher resolution grid that is local, i.e. it estimates short range forces on smaller independent sections that are hosted locally by individual processes. Thus, the distributed 3D FFTs are computed only on the coarse grid while the higher resolution grid computes highly efficient local FFTs.

This force splitting massively reduces the communication data-volume but at the cost of increasing the total number of operations performed. Thus while it scales better for large grids and highly distributed meshes, it can be excessive for small simulations. Therefore in `FlowPM` we implement both, a multi-grid scheme for large simulations, and the usual single-grid force scheme for small grid sizes, with the user having the freedom to choose between the two.

3.1. Multiresolution Pyramids

In this section we briefly discuss multiresolution pyramids that will later form the basis our multi-grid implementation for force estimation. As used in image processing and signal processing communities, image pyramids refer to multi-scale representations of signals and image built through recursive reduction operations on the original image (Burt and Adelson, 1983; Anderson et al., 1984). The reduction operation is a combination of - i) smoothing operation, often implemented by convolving the original image with a low-pass filter ($\mathcal{G}$) to avoid aliasing effects by reducing the Nyquist frequency of the signal, and ii) subsampling or downsampling operation ($\mathcal{S}$), usually by a factor of 2 along each dimensions, to compress the data. This operation can be repeatedly applied at every 'level' to generate a 'pyramid' structure of the low-pass filtered copies of the original image. Given a level g_l of original image g_0, the pyramid next level is generated by

$$g_{l+1} = \mathrm{REDUCE}(g_l) = \mathcal{S}[\mathcal{G} \star g_l] \tag{6}$$

where $\star$ represents a convolution. If the filter $\mathcal{G}$ used is a Gaussian filter, the pyramid is called a Gaussian pyramid.

However, in our multi-grid force estimation, we are interested in decoupling the large and small scale forces. This decoupling can be achieved by building upon the multi-scale representation of Gaussian pyramids to construct independent band-pass filters with a 'Laplacian' pyramid[1]. In this case, one reverses the reduce operation by - i) upsampling ($\mathcal{U}$) the image at level g_{l+1}, generally by inserting zeros between pixels and then ii) interpolating the missing values by convolving it with the same filter $\mathcal{G}$ to create the expanded low-pass image g'_l

$$g'_l = \mathrm{EXPAND}(g_{l+1}) = \mathcal{G} \star \mathcal{U}[g_{l+1}] \tag{7}$$

[1] Since we will not use the Gaussian kernel for smoothing, our pyramid structure is not a Laplacian pyramid in the strictest sense, but the underlying idea is the same.

A Laplacian representation at any level can then be constructed by taking the difference of the original and expanded low-pass filtered image at that level

$$L_l = g_l - g'_l \tag{8}$$

In Figure 1, we show the Gaussian and Laplacian pyramid constructed for an example image, along with the flow-chart for these operations.

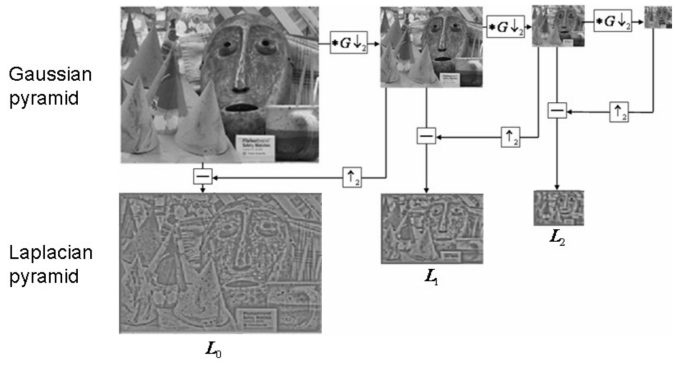

Figure 1: An example of Gaussian image pyramid and the corresponding Laplacian pyramid. The arrows and small boxes mark the operations involved[2].

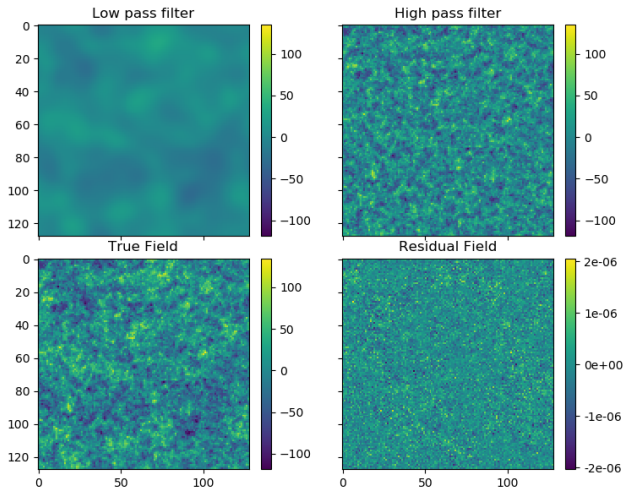

Figure 2: Laplacian pyramid for matter density field - the top row represents the upsampled low-pass filtered field and the high-pass filter field generated by taking the difference with the true field (bottom left). Residuals (bottom right) demonstrate that the original field is reconstructed perfectly with low and high-pass filter fields. The results are for a 400 Mpc/h box with 128^3 grid at high resolution and the coarse grid generated with low-pass filter before upsampling again is at 32^3.

Lastly, given the image at the highest level of the pyramid (g_N) and the Laplacian representation at every level ($L_0 ... L_{N-1}$), we can exactly recover the original image by recursively doing the series of following inverse transform-

$$g_l = L_l + \text{EXPAND}(g_{l+1}) \quad \forall l \in [N-1 ... 0] \tag{9}$$

[2]Image taken from http://graphics.cs.cmu.edu/courses/15-463/2012_fall/hw/proj2g-eulerian/

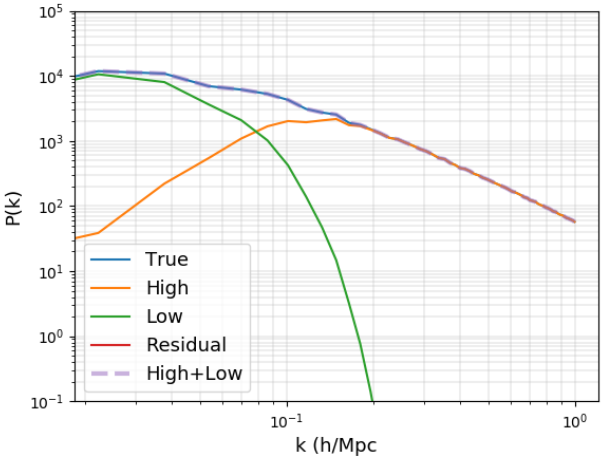

Figure 3: The power spectrum corresponding to the density fields shown in Figure 2. The red line is outside the scale for the residual field has negligible power, as expected.

In Figure 2, we show the resulting fields when these operations are applied on the matter density field. We show the result of upsampling the low-pass filtered image and the high-pass filtered image with the low-pass filtered component removed as is done in Laplacian pyramid. The negligible residuals show that the reconstruction of the fields is exact. In Figure 3, we show the power spectra of the corresponding fields to demonstrate which scales are contributed at every level. We will apply these operations in Section 5.2 to estimate forces on a two-level grid.

4. Mesh-TensorFlow

The last ingredient we need to build `FlowPM` is a model parallelism framework that is capable of distributing the N-body simulation over multiple processes. To this end, we adopt Mesh-TensorFlow (Shazeer et al., 2018) as the framework to implement our PM solver. Mesh-TensorFlow is a framework for distributed deep learning capable of specifying a general class of distributed tensor computations. In that spirit, it goes beyond the simple batch splitting of data parallelism and generalizes to be able to split any tensor and associated computations across any dimension on a computational mesh of processors.

We only summarize the key component elements of Mesh-TensorFlow here and refer the reader to Shazeer et al. (2018) [3] for more details. These are:

- a *mesh*, which is a n-dimensional array of processors with *named* dimensions.
- the dimensions of every tensor are also *named*. Depending on their name, these dimensions are either distributed/split across or replicated on all processors in a mesh. Hence every processor holds a *slice* of every tensor. The dimensions can thus be seen as 'logical' dimensions which will be split in the same manner for all tensors and operations.

[3]https://github.com/tensorflow/mesh

- a user specified *computational layout* to map which tensor dimension is split along which mesh dimension. The unspecified tensor dimensions are replicated on all processes. While the layout does not affect the results, it can affect the performance of the code.

The user builds a Mesh-TensorFlow graph of computations in Python. This graph is then *lowered*, which corresponds to transforming the symbolic Mesh TensorFlow graph into concrete tensors and operations to be located on individual GPUs of the mesh. It is at this stage that the defined tensor dimensions and operations are split amongst different processes on the mesh based on the computational layout. Mesh TensorFlow supports several implementations of the computation mesh onto physical devices. On Google's TPUs (Tensor Processing Units), Mesh TensorFlow provides a highly efficient parallel implementation, referred to as *SimdMeshImpl*, based on an SIMD paradigm (Single Instruction Multiple Devices). In this approach, TensorFlow operations (and communication collectives) are emitted from the perspective of one TPU, and this same program runs on every TPU, relying on the fact that each core actually performs the same operations. However, Mesh TensorFlow does not currently provide a similar SPMD (Single Program Multiple Data) implementation for GPU/CPU clusters, and instead relies on TensorFlow's native ability to incorporate several GPUs into the same graph over network. In this approach, referred to as *PlacementMeshImpl*, Mesh TensorFlow will rely on a single TensorFlow process to run the lowered computational graph, which can contain many GPUs. The disadvantage of this approach is that as the number of GPUs increase, so does the size of the graph managed by the single process, this translates for more than 16 GPUs into prohibitively long lowering times (i.e. time required to create all the concrete tensors and operations for each individual GPU). A new GPU SPMD mesh implementation is under active development and will ultimately alleviate these scaling issues for GPU clusters.

Returning to Mesh-TF computations, at the last step, the Mesh-TF tensors to be evaluated are *exported* to TF tensors in which case they are gathered on the single process.

Figure 4 shows an example code to illustrate these operations which form the starting elements of any Mesh-TensorFlow. The code also sets up context for FlowPM, naming the three directions of the PM simulation grid and explicitly specifying which direction is split along which mesh-dimension. Note that for a batch of simulations, computationally the most efficient splitting will almost always maximize the splits along the batch dimension. While `FlowPM` supports that splitting, we will henceforth fix the batch size to 1 since our focus is on model parallelism of large scale simulations.

5. `FlowPM`: `FastPM` implementation in Mesh-Tensorflow

Finally, in this section, we introduce `FlowPM` and discuss technical details about its implementation in Mesh-TensorFlow. We have already discussed the basic operations of the underlying PM scheme implemented - the generation of initial conditions, kick and drift operations, force estimation and associated

```python
import mesh_tensorflow as mtf
import tensorflow.compat.v1 as tf

# Setup graph and associated mesh
graph = mtf.Graph()
mesh = mtf.Mesh(graph, "my_mesh")

# Define named dimensions for defining a 3D volume
x_dim, y_dim, z_dim = (mtf.Dimension("ncx", 128),
                       mtf.Dimension("ncy", 128),
                       mtf.Dimension("ncz", 128))
batch_dim = mtf.Dimension("batch", 1)

# Sample a batched random normal 3D volume
field = mtf.random_normal(mesh,
        [batch_dim, x_dim, y_dim, z_dim])

# [...] Other mtf operations can be added here

# Define a concrete implementation as a 2D mesh on 8 GPUs,
# splitting `ncx` and `ncy` dimensions along rows and cols
mesh_impl = mtf.placement_mesh_impl.PlacementMeshImpl(
        mesh_shape=[("row", 4), ("col", 2)],
        layout_rules=[("ncx", "row"), ("ncy", "col")],
        devices=["device:GPU:%d"%i for i in range(8)])

#Lower the graph onto computational mesh
lowering = mtf.Lowering(graph, {mesh:mesh_impl})
# Retrieve mtf tensor as TensorFlow tensor
tf_field = lowering.export_to_tf_tensor(field)

# Evaluate graph
with tf.Session as sess:
    sess.run(tf_field)
```

Figure 4: Sample code to generate random normal field of grid size 128 on a 2D computational mesh of 8 GPUs with x and y directions split across 4 and 2 processors respectively

kernels as modified in `FastPM` - in Section 2. In this section, we focus on the two aspects of `FlowPM` which are done differently, the domain decomposition and the multi-grid scheme for force estimation, the latter of which is novel to `FlowPM`.

5.1. Domain decomposition and Halo Exchange

In Mesh-Tensorflow, every process has a slice of every tensor. Thus for our PM grid, where we split the underlying grid spatially along different dimensions, every process will hold only the physical region and the particles corresponding to that physical region. However at every PM step, there is a possibility of particles moving in-and-out of any given slice of the physical region. At the same time, convolution operations on any slice need access to the field outside the slice when operating on the boundary regions. The strategy for communication between neighboring slices to facilitate these operations is called a *halo-exchange*. We implement this exchange by padding every slice with buffer regions of *halo size* (h) that are shared between the slices on adjacent processes along the mesh. The choice of *halo size* depends on two factors[4], apart from the increased

[4] In case of the pyramid scheme, the halo size is defined on the high resolution grid and we reduce it by the corresponding downsampling factor for the coarse grid.

memory cost. Firstly, the halo size should be large enough to ensure an accurate computation of smoothing operations, i.e. larger than half the smoothing kernel size. Secondly, to keep communications and book-keeping to a minimum, in the current implementation particles are not transferred to neighbouring processes if they travel outside of the domain of a given process. This means that we require halo regions to be larger than the expected maximum displacement of a particle over the entire period of the simulation. In our tests, we find that the currently used 50 Mpc/h halo size works well across resolutions (from 0.5 Mpc/h-10 Mpc/h grid resolution) for cold dark matter simulations.

Our halo-exchange algorithm is outlined in Algorithm 1 and illustrated for a 1-dimension grid in Figure 5. It is primarily motivated by the fact that Mesh-TensorFlow allows to implement independent, local operations simply and efficiently as *slice-wise* operations along the dimensions that are not distributed. Thus we can implement all of the operations - convolutions, CIC interpolations, position and velocity updates for particles and local FFTs - on these un-split dimensions as one would for a typical single-process implementation without worrying about distributed strategies. Similarly, the distributed FFTs are also implemented directly as Mesh-TF operations, without using any external libraries. To take advantage of this, we begin by reshaping our tensors from 3 dimensions (excluding batch dimension) to 6 dimensions such that the first three indices split the tensor according the computational layout and the last three indices are not split, but replicated across all the processes and correspond to the local slice of the grid on each process. This is shown in Figure 5, where an original 1-D tensor of size $N = 12$ in split over $nx = 2$ processes. In the second panel, this is reshaped to a 2-D tensor with the first dimension $nx = 2$ split over processes and s_x is the un-split dimension. The un-split dimension is then zero-padded with *halo size* resulting in the size along the un-split dimension to be $s_x = N/n_x + 2h$. This is followed by an exchange with the neighboring slices over the physical volume common to the two adjacent processes.

After the slicewise operation updates the local slices, the same steps are followed in reverse. The padded halo regions are exchanged again to combine updates in the overlapping volume from adjacent processes. Then the padded regions are dropped and the tensor is reshaped to the original shape.

5.2. Multi-grid Scheme with Multiresolution Pyramids

To implement the multi-grid scheme with multiresolution pyramids, we need a smoothing and a subsampling operation. While traditionally smoothing operations are performed in Fourier space in cosmology, here we are restricted to these operations in local, pixel space. However this leads to a tradeoff between the size of the smoothing kernel in pixel space, and how compact the corresponding low pass filter is in Fourier space. Our main consideration is to ensure that the low-resolution component of the pyramid is sufficiently suppressed at half the Nyquist scale of the original field, so as to be able to downsample this low resolution field with minimum aliasing. Using larger kernels improves the Fourier drop-off but implies an in-

Algorithm 1 Halo Exchange

global variables
N - size of the global grid
h - halo size, to be padded
nx, ny, nz - #splits along three directions
sx, sy, sz - size of local slice after split
nx, ny, nz - Name of dimension along nx, ny, nz
sx, sy, sz - Name of dimension along sx, sy, sz
end global variables

procedure RESHAPE_EXPAND(G)
ASSERT(G.shape == (N, N, N))
$G \Leftarrow G$.reshape(nx, ny, nz, sx, sy, sz)
for $axis \in$ sx, sy, sz **do**
$G \Leftarrow$ ZEROPAD($G, size = h, axis = axis$)
end for
return G
end procedure

procedure RESHAPE_REDUCE(G)
ASSERT(G.shape == $(nx, ny, nz, sx + 2h, sy + 2h, sz + 2h)$)
for $axis \in$ sx, sy, sz **do**
$G \Leftarrow$ REMOVEPAD($G, size = h, axis = axis$)
end for
$G \Leftarrow G$.reshape(N, N, N)
return G
end procedure

procedure EXCHANGE(G)
Add updates in overlapping padded regions from neighboring processes
end procedure

procedure HALOEXCHANGEOP($G, SlicewiseOp$)
▷ Strategy for operating slicewise op with halo exchange
ASSERT(G.shape == (N, N, N))
$G \Leftarrow$ RESHAPE_EXPAND(G)
$G \Leftarrow$ EXCHANGE(G)
$G \Leftarrow$ SLICEWISEOP(G)
$G \Leftarrow$ EXCHANGE(G)
$G \Leftarrow$ RESHAPE_REDUCE(G)
return G
end procedure

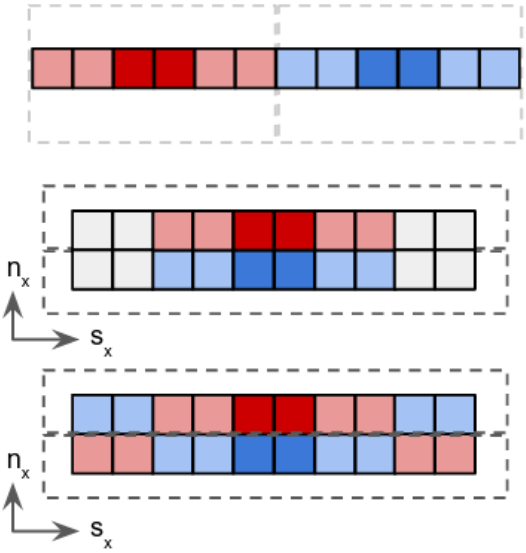

Figure 5: An illustration of the halo exchange strategy as outlined in Algorithm 1 for a simple 1-D tensor of size $N = 12$ and *halo size* $h = 2$, specifically Reshape_Expand operation. The red and blue regions are to be split on two process, with light regions indicating the overlapping volume that will be exchanged. In the second panel, the tensor is reshaped with the split along the first, nx dimension and the second un-split s_x that is zero-padded. In the third panel, the regions are exchanged (here under the assumption of periodic boundary conditions).

creased computational and memory cost of halo-exchanges to be performed at the boundaries.

In our experiments, we find that a 3-D bspline kernel of order 6 balances these trade-offs well and allows us to achieve sub-percent accuracy, as we show later. Furthermore in `FlowPM`, we take advantage of highly efficient TensorFlow operations and implement both, the smoothing and subsampling operations together by constructing a 3D convolution filter with this bspline kernel and convolving the underlying field with convolutions with stride 2 i.e. the convolution filter is moved by 2 grid points. In our default implementation, we repeat these operations twice and our global grid is 4x coarser than the higher resolution grid, resulting in 64x reduction in volume of communication for global FFTs. The full algorithm for this REDUCE operation is outlined in Algorithm 2. Based on the discussion in Section 3.1, we also outline the reverse operation with EXPAND method.

Finally we are in the position to outline the multi-grid scheme for force computation. This is outlined in Algorithm 3. Schematically, we begin by reducing the high resolution grid to generate the coarse grid. Then we estimate the force components in all three directions on both the grids, with the key difference that the coarse grid performs distributed FFT on a global grid while for the high resolution grid, every process performs local FFTs in parallel on the locally available grids. To recover the correct total force, we need to combine these long and the short range forces together. For this, we expand the long-range force grid back to the high resolution. At

Algorithm 2 Methods for pyramid scheme

procedure REDUCE(H, $f = 2$, $n = 6$, $S = 2$)
▷ Downsample field H by convolving with a bspline kernel of order n consecutively f times with stride S
$K \Leftarrow$ BSpline(n)
$D \Leftarrow H$
for $i \in 0 \ldots f$ **do**
$D \Leftarrow$ Conv3D(D, K, S)
end for
return D
end procedure

procedure EXPAND(D, $f = 2$, $n = 6$, $S = 2$)
▷ Upsample field D by convolving with a bspline kernel of order n consecutively f times with stride S
$K \Leftarrow$ BSpline(n) x 8
$H \Leftarrow D$
for $i \in 0 \ldots f$ **do**
$H \Leftarrow$ TransposedConv3D(H, K, S)
end for
return H
end procedure

the same time, we remove the long-range component from the high-resolution grid with a reduce and expand operation, similar to the band-pass level generated in the Laplacian pyramid in Section 3.1. In the end, we combine these long and short range forces to recover the original force. We compare this with the particle-mesh forces evaluated on a single grid in Figure 6 and show that we recover the original force accurately across all scales without any fine-tuning.

Note that in implementing a multi-grid scheme for PM evolution, only the force calculation in the kick step is modified, while the rest of the PM scheme remains the same and operates on the original high-resolution grid.

6. Scaling and Numerical Accuracy

In this section, we will discuss the numerical accuracy and scaling of these simulations. Since the novel aspects of `FlowPM` are a differentiable Mesh-Tensorflow implementation and multigrid PM scheme, our discussion will center around comparison with the corresponding differentiable Python implementation of `FastPM` which shares the same underlying PM scheme. We do not discuss the accuracy with respect to high resolution N-Body simulations with correct dynamics, we touch upon this in Section 2 and refer the reader to Feng et al. (2016) for further details on such a comparison.

Our tests are performed primarily on GPU nodes on Cori supercomputer at NERSC, with each node hosting 8 NVIDIA V100 ('Volta') GPUs each with 16 GB HBM2 memory and connected with NVLink interconnect. We compare our results and scaling with the differentiable Python code run on Cori-Haswell nodes. This consists of `FastPM` code for forward mod-

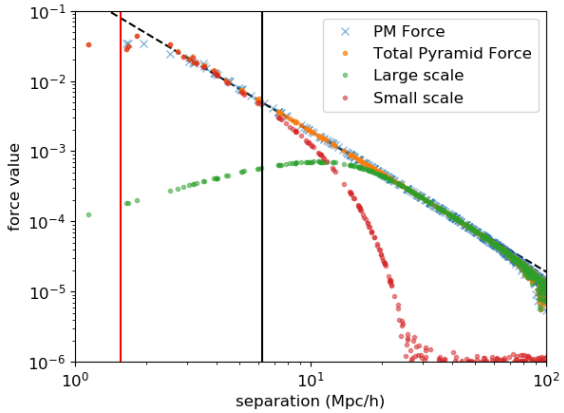

Figure 6: Comparing the accuracy of PM forces with the multi-grid forces for a 128^3 grid, 200 Mpc/h box wherein the x-direction is split over 2 GPU. The small and large scale forces estimated with multi-grid scheme add to give the total force in the pyramid implementation. Red and black vertical lines are the grid resolutions of the high resolution (also the original resolution) and coarse grids respectively.

eling, `vmad`[5] for automated differentiation and `abopt`[6] for optimization.

To distinguish between a single grid and multi-grid implementation in the subsequent text, we will refer to the former as *mesh* scheme and the latter as the *pyramid* scheme.

6.1. Accuracy of the simulations

We begin by establishing the accuracy of `FlowPM` for both mesh and the pyramid scheme with the corresponding `FastPM` simulation. We run both the simulations in a 400 Mpc/h box on 128^3 grid for 10 steps from $z = 9$ to $z = 0$ with force-resolution of 1. The initial conditions are generated at the 2nd order in Lagrangian perturbation theory (LPT). The `FastPM` simulation is run on a single process while the `FlowPM` simulations are run on different mesh-splits to validate both the multi-process implementations. We use halo size of 16 grids cells, corresponding to a physical size of 50 Mpc/h in the current experiments, and find that increasing the halo size increases the accuracy marginally for all configurations. Moreover, given the more stringent memory constraints of GPU, we run the `FastPM` simulation with a 64-bit precision while the `FlowPM` simulations are run with 32-bit precision.

Figure 7 compares the two simulations at the level of the fields. For the `FlowPM` simulations, we show the configuration run on 4 GPUs, with the grid split in 2 along the x and y direction each. We find that both, the single grid and the pyramid implementation are accurate with sub-percent residuals at the field level.

In Figure 8 we compare the two simulations more quantitatively by measuring their clustering properties. To compare

[5] https://github.com/rainwoodman/vmad
[6] https://github.com/bccp/abopt

Algorithm 3 Force computation in Muti-grid pyramid method

procedure ForceGrids($G, mode$)
  ▷ Estimate the component force grids with 3D FFT
  $\tilde{G} \Leftarrow$ FFT3D($G, mode$)
  for $i \in x, y, x$ **do**
    $F_i \Leftarrow$ iFFT3D($\nabla_i \nabla^{-2} \tilde{G}, mode$)
  end for
  return $[F_x, F_y, F_z]$
end procedure

procedure Force($H,\ f = 2,\ n = 6,\ S = 2$)
  ▷ Pyramid scheme to estimate force meshes
  $D \Leftarrow$ REDUCE($H,\ f,\ n,\ S$)
  $F_L \Leftarrow$ ForceGrids($D, mode = distributed$)
  $F_S \Leftarrow$ ForceGrids($H, mode = local$)
  for $i \in x, y, x$ **do**
    $F_{i,L} \Leftarrow$ EXPAND($F_{i,L},\ f,\ n,\ S$)
    $F_{i,l} \Leftarrow$ REDUCE($F_{i,S},\ f,\ n,\ S$)
    $F_{i,l} \Leftarrow$ EXPAND($F_{i,l},\ f,\ n,\ S$)
    $F_{i,S} \Leftarrow F_{i,S} - F_{i,l}$
    $F_i \Leftarrow F_{i,L} + F_{i,S}$
  end for
  return $[F_x, F_y, F_z]$
end procedure

their 2-point functions, we measure the transfer function which is the ratio of the power spectra ($P(k)$) of two fields

$$T_f(k) = \sqrt{\frac{P_a(k)}{P_b(k)}} \tag{10}$$

and their cross correlation which compares the phases and hence is a measure of higher order clustering

$$r_c(k) = \frac{P_{ab}(k)}{\sqrt{P_a(k) \times P_b(k)}} \tag{11}$$

with P_{ab} being the cross power spectra of the two fields.

Both the transfer function and cross correlation are within 0.01% across all scales, with the former starting to deviate marginally on small scales. Thus the choices made in terms of convolution filters for up-down sampling, halo size and other parameters in multi-grid scheme, though not extensively explored, are adequate to reach the requisite accuracy. We anticipate this to be a grid resolution dependent statement, and the resolution we have chosen is fairly typical of cosmological simulations that will be run for the analysis of future cosmological surveys and the analytic methods that will most likely use these differentiable simulations.

6.2. Scaling on Cori GPU

We perform the scaling tests on the Cori GPUs for varying grid and mesh sizes, and compare it against the scaling of the Python implementation of `FastPM`.

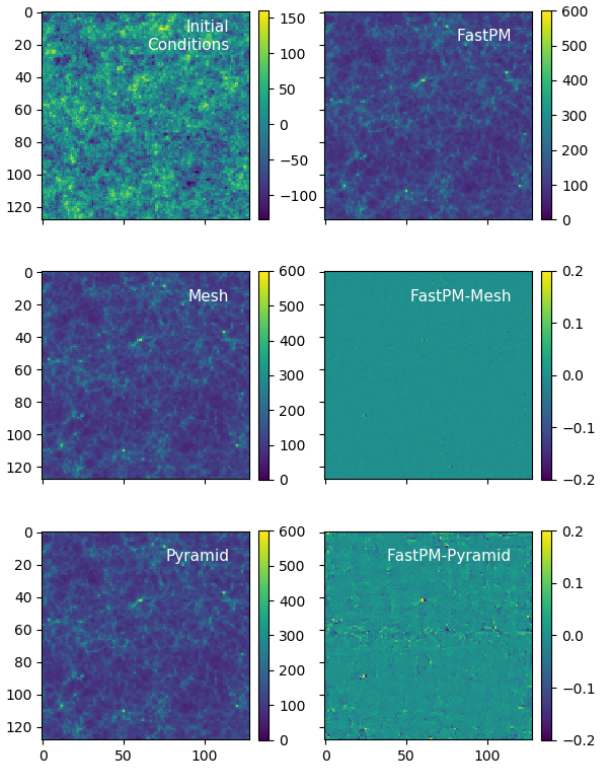

Figure 7: Comparison of the accuracy of `FlowPM` simulation for Mesh (second left) and Pyramid (third left) scheme with `FastPM` simulation (top right) for the same initial conditions at the level of fields. The residuals are shown in second and third right image. Configuration is 10 step simulation with 128^3 grid and 400 Mpc/h in size. `FastPM` simulation is run on single process while `FlowPM` simulations are run on 4 processes with nx=2 and ny=2, the splits in x and y direction. Third axis is summed over the box for the projections.

6.2.1. *FFT Scaling*

Since 3D FFTs are typically the most computationally expensive part of a PM simulation, we begin with their time-scaling. For the Mesh-TF implementation, where we have implemented these operations as low-level Mesh-TensorFlow operators. We show the time scaling for these operations in Figure 9. For small grids, the timing for FFT is almost completely dominated by the time spent in all-to-all communications for transpose operations. As a result, the scaling is poor with the timing increasing as we increase the number of processes. However for very large grids, i.e. $N \geq 512$, the compute cost approaches communication cost for small number of process and we see a slight dip up to 8 processes, which is the number of GPUs on a single node. However after that, further increase in number of processes leads to a significant increase in time due to inter-node communications. We show the results only for balanced splits in mesh layout in x and y directions i.e. where the x and y directions are split in equally or comparably since these lead to the best timings. Unbalanced splits lead to worse performance than a balanced split, with the difference in timings becoming larger for large grid sizes.

We compare the timing of FFT with that implemented in

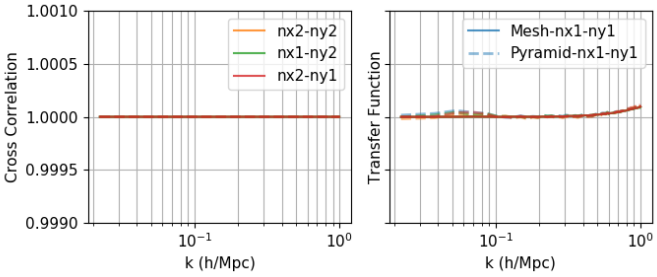

Figure 8: Compare the accuracy of `FlowPM` simulation for single mesh (solid) and Pyramid (dashed lines) scheme with `FastPM` simulation at the level of 2 point functions, cross-correlation (left) and transfer-function (right). Configuration is 5 step simulation with 128^3 grid and 400 Mpc/h in size. `FastPM` simulation is run on single process while `FlowPM` simulations are run on different number of processes with nx and ny as splits in x and y direction respectively, as indicated in the legends with different colors. The legends are shared between two subplots the lines not distinctly seen are overlapping.

`FastPM` which implements the PFFT algorithm for 3D FFTs (Pippig, 2013). The scaling for PFFT is close to (inversely) linear with the increasing number of processes. However due to GPU accelerators, the Mesh-TF FFTs are still orders of magnitudes faster in most cases, with only getting comparable for small grid sizes and large mesh sizes.

6.2.2. *PM Step Scaling*

While the scaling of FFTs is sub-optimal, what is more relevant for us is the scaling of 1 PM step that involves other operations in addition to FFTs. Thus, next we turn to the scaling of generating initial conditions (ICs) for the cosmological simulation. It provides a natural test since this step involves all the operations that enter a single PM step i.e. kick, drift, force evaluation and interpolation. To estimate the scaling for an entire simulation, a good approximation is simply multiplying the time taken for this step with the number of time-steps in the simulation. In this experiment, our ICs are generated as first-order LPT displacements i.e. Zeldovich displacements Zel'Dovich (1970) (though the default code for the simulations uses second order displacements). Schematically, the operations involved in this are outlined in Algorithm 4.

Algorithm 4 Generating IC

procedure GEN-IC(N, pk)
  $g \Leftarrow$ sample 3D normal random field of size N
  $\delta_L \Leftarrow$ Scale g by power spectrum pk
  $F \Leftarrow$ Estimate force at grid-points from δ_L i.e. Force
  $d \Leftarrow$ Scale F to obtain displacement
  $X \Leftarrow$ Displace particles with d, i.e. Drift
  $V \Leftarrow$ Scale to obtain velocity of particles i.e. Kick
  $\delta \Leftarrow$ Interpolate particles at X to obtain density
  return δ
end procedure

We establish the scaling for this step in Figure 10 for both the implementations in `FlowPM`- the single grid as well as pyramid

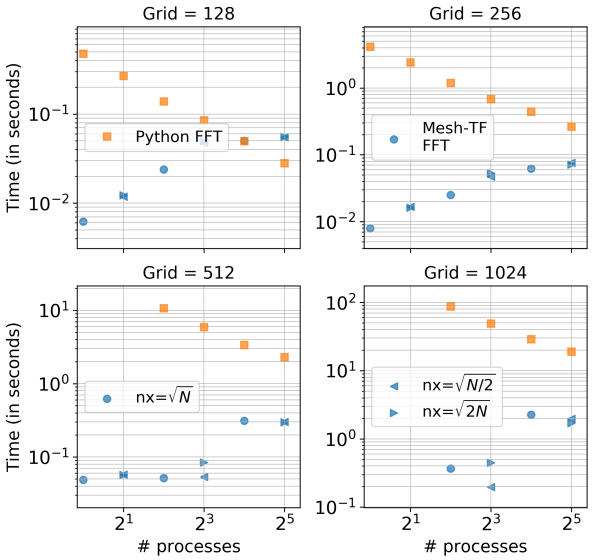

Figure 9: Compare scaling for a single forward+backward 3D FFT implemented in `FlowPM` in Mesh-Tensorflow, with PFFT implementation in Python-`FastPM` for different grid sizes and number of processes. For Mesh-TF implementation, we show only the balanced layouts where the number of splits nx and ny in the two directions is the same (circles) or comparable (arrows), depending on the total number of processes (N). In ideal scaling, wall clock time is inversely proportional to the number of processes.

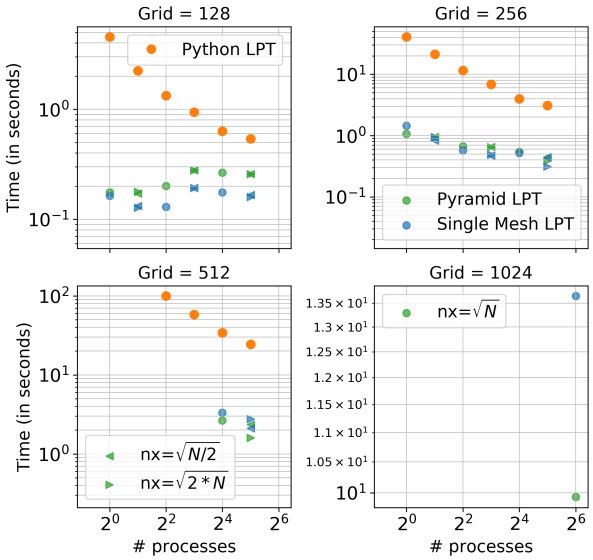

Figure 10: Compare the scaling for generating initial conditions with 1st order LPT in Mesh-Tensorflow with Python-`FastPM` for different grid sizes and number of processes. We show the scaling for both, single mesh implementation (blue) and pyramid implementation (green) in `FlowPM` in Mesh-Tensorflow. Missing points in expected configurations, for example 1024 grid in python implementation, run out of memory.

scheme. Firstly, note that since this step combines computationally intensive operations (like interpolation) with communication intensive operations (like FFT), the scaling for a PM step generally has the desirable slope with the timings improving as we increase the number of processes. In fact, we find that roughly equal time is spent in FFT operations as in gather operation, which collects the interpolates the particles on the grid points to mass density for force calculation, as well as the scatter operation which gathers the force from grid points back to the particle positions. Updating particle positions and velocities take negligible time in comparison. There is, however, still a communication overhead as we move from GPUs on a single node to multi-nodes (see for grid size of 128). Secondly, as expected, given the increase in number of operations for the pyramid scheme as compared to single-mesh implementation the single-mesh implementation is more efficient for small grids. However its scaling is poorer than the pyramid scheme which is $\sim 20\%$ faster for grid size of $N = 1024$.

In Figure 10, we also compare this implementation with the Python implementation. We find that `FlowPM` is atleast 10x faster than `FastPM` for the same number of processes, across all sizes (except the combination of smallest grid and largest mesh size). Moreover, since the both `FlowPM` and `FastPM` benefit with increasing the number of processes, we expect that it will be hard to beat the performance of `FlowPM` with simply increasing the number of processes in `FastPM`.

7. Example application: Reconstruction of initial conditions

With the turn of the decade, the next generation of cosmological surveys will probe increasingly smaller scales, over the largest volume available, with different cosmological probes. As a result, there is a renewed interest in developing forward modeling approaches (Jasche and Wandelt, 2013; Seljak et al., 2017; Elsner et al., 2019) and simulations based inference techniques to optimally extract and combine information from these surveys. Differentiable simulators such as `FlowPM` will make these approaches tractable in cosmology, especially in the challenging regime of big data volumes of future surveys, and hence play an important role in their analysis.

Here we demonstrate the efficacy of `FlowPM` with one such analytic approach that has recently received a lot of attention. We are interested in reconstructing the initial conditions of the Universe (**s**) from the late-time observations (**d**) (Wang et al., 2014; Jasche and Wandelt, 2013; Seljak et al., 2017; Modi et al., 2018; Schmittfull et al., 2018). The late time matter-density field is highly non-Gaussian which makes it hard to model and do inference. Hence the current analysis methods only extract a fraction of the available information from cosmological surveys. On the other hand, the initial conditions are known to be Gaussian to a very high degree and hence their power spectrum (**S**) is a sufficient statistic. Thus reconstructing this initial density field provides a way of developing optimal approach for inference (Seljak et al., 2017). At the same time, the reconstructed initial field can be forward modeled to generate other latent cosmological fields of interest that can be used for supple-

mentary analysis (Modi et al., 2019b; Horowitz et al., 2019a).

We approach this reconstruction in a forward model Bayesian framework based most directly on Seljak et al. (2017); Modi et al. (2018, 2019b). We refer the reader to these for technical details but briefly, we forward model ($\mathcal{F}$) the observations of interest from the initial conditions of the Universe ($\mathbf{s}$) and compare them with the observed data ($\mathbf{d}$) under a likelihood model - $\mathcal{L}(\mathbf{d}|\mathbf{s})$. This can be combined with the Gaussian prior on the initial modes to write the corresponding posterior - $\mathcal{P}(\mathbf{s}|\mathbf{d})$. This posterior is either optimized to get a maximum-a-posteriori (MAP) estimate of the initial conditions, or alternatively it can be explored with sampling methods as in Jasche and Wandelt (2013). However in either approach, given the multi-million dimensionality of the observations, it is necessary to use gradient-based approaches for optimization and sampling and hence a differentiable forward model is necessary. Particle-mesh simulations evolving the initial conditions to the final matter field will form the first part of these forward models for most cosmological observables of interest. Here we demonstrate in action with two examples how the inbuilt differentiability and interfacing of `FlowPM` with TensorFlow makes developing such approaches natural.

7.1. Reconstruction from Dark Matter

In the first toy problem, consider the reconstruction of the initial matter density field ($\mathbf{s}$) from the final Eulerian dark matter density field ($\mathbf{d} = \delta_e$) as an observable. While the 3D Eulerian dark matter field is not observed directly in any survey, and weak lensing surveys only allow us to probe the projected matter density field, this problem is illustrative in that the forward model is only the PM simulation i.e. $\mathcal{F}$ = FlowPM with the modeled density field $\delta_m = \mathcal{F}(\mathbf{s})$. We will include more realistic observation and complex modeling in the next example.

We assume a Gaussian uncorrelated data noise with constant variance σ^2 in configuration space. Then, the negative log-posterior of the initial conditions is:

$$-\log p(\delta_m|\delta_e) = \frac{(\delta_e - \delta_m)^2}{2\sigma^2} + \frac{1}{2}\mathbf{s}^\dagger\mathbf{S}^{-1}\mathbf{s} + cst \qquad (12)$$

where the first term is the negative Gaussian log-likelihood and the second term is the negative Gaussian log-prior on the power spectrum $\mathbf{s}$ of initial conditions, assuming a fiducial power spectrum $\mathbf{S}$.

To reconstruct the initial conditions, we need to minimize the negative log-posterior Eq. 12. The snippet of Mesh-TensorFlow code for such a reconstruction using TF Estimators (Cheng et al., 2017) is outlined in Listing 11. In the interest of brevity, we have skipped variable names and dimension declarations, data I/O and other *setup code* while focusing only on the logic directly relevant to reconstruction. Broadly, we use the Estimators class defined TensorFlow to implement reconstruction as an optimization problem where the initial density field is treated as the underlying *variable* and the negative log-posterior is the loss function of interest. Using Estimators allows us to naturally reuse the entire machinery of TensorFlow including but not limited to monitoring optimization, choosing from various inbuilt optimization algorithms, restarting optimization from checkpoints and others. Thematically, the entire code can be split into three components:

- recon_model : this generates the computational graph for the optimization problem. It creates a *variable* linear field (initial conditions) that is the input to the forward model, which is `FlowPM`, to generate our model that is compared with the input data to construct a likelihood and prior term, hence defining the negative log-posterior which is the loss metric to optimize for reconstruction

```
def recon_model(mesh, data, x0):
    # [...] some code initialisation
    #Define initial conditions as variable
    var=mtf.get_variable(mesh,
                         'linear',
                         shape,
                         tf.constant_initializer(x0))
    #Forward model here with FlowPM
    model=FLOWPM(var, ...)
    #Define metrics for loss with data noise variance
    #"sigma" and prior power spectrum "power"
    chisq=mtf.reduce_sum(((model-data)/sigma)^2)
    prior=mtf.reduce_sum(r2c(var)^2/power)
    loss =chisq + prior
    return var, model, loss

def model_fn(x0, data, mode, params):
    # [...] some code initialisation
    var, model, loss=recon_model(mesh, data, x0)
    #Construct optimizer for reconstruction
    if mode == tf.estimator.ModeKeys.TRAIN:
        var_grads=mtf.gradients([loss],[v.outputs
        for v in graph.trainable_variables])
        optimizer=mtf.optimize.AdamOptimizer(0.1)
        update_ops=optimizer.apply_grads(var_grads,
                         graph.trainable_variables)
    #Lower mesh tensorflow variables and ops
    lowering=mtf.Lowering(graph,{mesh:mesh_impl})
    tf_init=lowering.export_to_tf_tensor(var)
    tf_model=lowering.export_to_tf_tensor(model)
    tf_loss=lowering.export_to_tf_tensor(loss)
    #If predict, return current reconstruction
    if mode == tf.estimator.ModeKeys.PREDICT:
        tf.summary.scalar("loss", tf_loss)
        predictions={"ic":tf_init,"data":tf_data}
        return tf.estimator.EstimatorSpec(
          mode=tf.estimator.ModeKeys.PREDICT,
          predictions=predictions)
    #If train, optimize for reconstruction
    if mode == tf.estimator.ModeKeys.TRAIN:
        tf_up_op=[lowering.lowered_operation(op)
                  for op in update_ops]
        train_op=tf.group(tf_up_op)
        # ...checkpoint hooks...
        return tf.estimator.EstimatorSpec(
                tf.estimator.ModeKeys.TRAIN,
                loss=tf_loss, train_op=train_op)

def main():
    #  [...] some code initialisation
    #Define estimator for reconstruction
    def input_fn():
        return x0, datafea
    recon_estimator = tf.estimator.Estimator(
            model_fn=model_fn,
            model_dir='./tmp/')
    # Train (Reconstruct)
    recon_estimator.train(input_fn, max_steps=100)
    #and evaluate model.
    eval_results = recon_estimator.predict(input_fn)["ic"]
```

Figure 11: (Listing 7.1) Code to reconstruct the initial conditions from the observed matter density field with `FlowPM` and TensorFlow Estimator API

- model_fn : this creates the train and predict routines for the TF Estimator. If the mode is training, it estimates the gradients of the objective function returned by the recon_model with respect to the variable defined therein, passes it to the optimzation algorithm and updates the variables. If the mode is predict, it takes in the current estimate of the underlying variable and returns the model prediction. As previously mentioned, in MeshTF, one requires to *lower* these tensors which converts them from abstract objects to objects which can be evaluated.
- main : this calls the Estimator class to first do reconstruction by running the estimator in training mode (to do optimization) and then in predict mode to evaluate our reconstruction

We show the results of this reconstruction in Figure 12 and 13. Here, the forward model is a 5-step PM simulation in a 400 Mpc/h box on 128^3 grid with force resolution of one i.e. the PM force is estimated on the same grid resolution as particle resolution (in this case 128^3. We supplement the gradient-based optimization outlined in Listing 11 with annealing scheme as described in Feng et al. (2018); Modi et al. (2018, 2019b) to assist reconstruction of large scales. Briefly, this involves smoothing the residual on small scales to fit the large scales first, and then we keep reducing the smoothing scale to fit increasingly smaller scales of the density field. We skip technical details here in the code for the sake of simplicity, but they are included straightforwardly in this API as a part of recon_model and training spec.

We compare our results with the previous reconstruction code in Python based on `FastPM`, `vmad` and `abopt` (henceforth referred to as `FastPM` reconstruction). Both the reconstructions follow the same five step annealing schedule to promote convergence on large scales as described in Modi et al. (2018); Feng et al. (2018). `FastPM` reconstruction uses gradient descent algorithm with single step line-search. Every iteration (including gradient evaluation and line search) takes roughly $\sim$ 4.6 seconds on 4 nodes i.e. 128 cores on Cori Haswell at NERSC. We reconstructed for 500 steps, until the large scales converged, in total wall-clock time of $\sim$ 2300 seconds. In comparison, for `FlowPM` reconstruction we take advantage of different algorithms in-built in TensorFlow and use Adam (Kingma and Ba, 2014) optimization with initial learning rate of 0.01, without any early stopping or tolerance. Every annealing step runs for 100 iterations wherein every iteration takes $\sim$ 1.1 seconds on a single Cori GPU and the 500 iterations done for total optimization clock in at $\sim$ 550 seconds in wallclock time. These numbers, except time per iteration, will likely change with a different learning rate and optimization algorithm and we have not explored these in detail for this toy example.

In Figure 12, we show the true initial and data field along with the reconstructed initial field by both codes. In Fig 13, we show the cross correlation and transfer function between the reconstructed and true initial fields (blue) as well as the final data field (orange) for `FlowPM`. Here the reconstructed final field for comparison with the final data field is obtained by forward modeling the reconstructed initial field. `FastPM` reconstructs the fields comparably, even though we use a different optimization algorithm.[7]

[7] `abopt` also provides the flexibility to use Limited-memory BFGS (L-

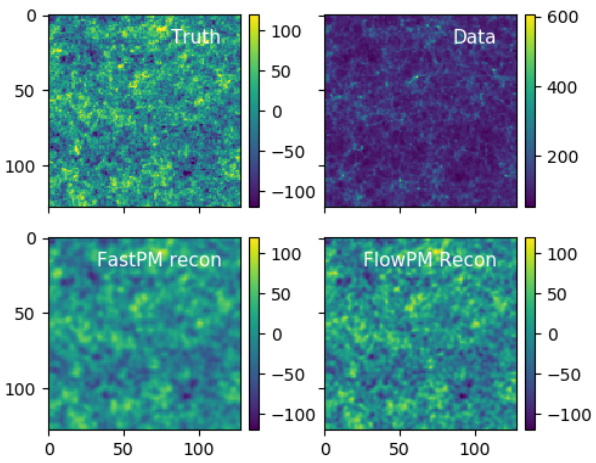

Figure 12: Reconstructing initial conditions from the dark matter field observable. Top row shows the true initial conditions (left) and the dark matter data (right). Bottom row shows the reconstructed MAP initial density field with `FastPM` and `FlowPM` respectively. The reconstructed fields and power is similar on large scales, which are the scales of primary interest, but different on small scales due to different optimizers being used in different codes. `FlowPM` reconstructs more power on small scales than `FastPM` and hence looks visually sharper.

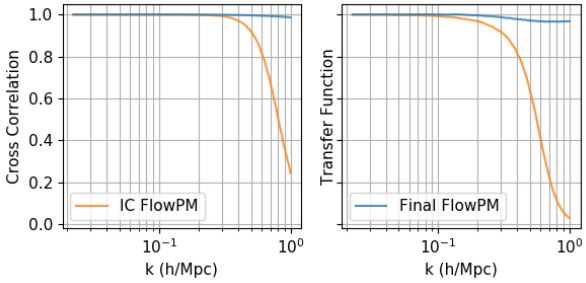

Figure 13: Two point statistics for the reconstructed fields from the dark matter field observable. We show the cross correlation coefficient (left) and transfer function (right) for the reconstructed initial field (orange) and the corresponding final dark matter field (blue) with `FlowPM`. The reconstruction is near perfect on the large scales.

7.2. Reconstruction from Halos with a Neural Network model

In the next example, we consider a realistic case of reconstruction of the initial conditions from dark matter halos under a more complex forward model. Dark matter halos and their masses are not observed themselves. However they are a good proxy for galaxies which are the primary tracers observed in large scale structure surveys, since they capture the challenges posed by galaxies as a discrete, sparse and biased tracer of the underlying matter field. Traditionally, halos are modeled as a biased version of the Eulerian or Lagrangian matter and associated density field (Sheth and Tormen, 1999; Matsubara, 2008; Schmittfull et al., 2018). To better capture the discrete nature of the observables at the field level, Modi et al. (2018) proposed using neural networks to model the halo masses and positions from the Eulerian matter density field. Here we replicate their formalism and reconstruct the initial conditions from the halo mass field as observable.

In Modi et al. (2018), the output of PM simulation is supplemented with two pre-trained fully connected networks, NNp and NNm. NNp is a fully connected network that predicts a binary mask on the grid with ones corresponding to the grid points where a halo is predicted to be present, and NNm predicts halo masses at every grid point. These are then combined to predict a halo-mass field (M_{NN}) that is compared with the observed halo-mass field (M_d) up to a pre-chosen number density. They propose a heuristic likelihood for the data inspired from the log-normal scatter of halo masses with respect to observed stellar luminosity. Specifically,

$$\mathcal{L} = \frac{\mu + \log(M_d^R + M_0) - \log(M_{NN}^R + M_0))}{2\sigma^2} \tag{13}$$

where M_0 is a constant varied over iterations to assist with optimization, M^R are mass-fields smoothed with a Gaussian of scale R and μ and σ are mass-dependent mean and standard deviations of the log-normal error model estimated from simulations.

The code snippet in Listing 14 modifies the code in 11 by including these pre-trained neural networks NNp and NNm in the forward model to supplement `FlowPM`. It also demonstrates how to include associated variables like M_0 in the recon_model function that can be updated with ease over iterations to modify the model and optimization.

In this experiment, we do reconstruction with a simulation data of the halo mass field with number density $\bar{n} = 10^{-3}$ (Mpc/h)$^{-3}$ on the 128^3 grid for a 400 Mpc/h box. At this resolution and number density, only 17.5% of grid points are non-zero, indicating the sparsity of our data. The forward model is a 5 step `FastPM` simulation followed by the two fully connected networks outlined above, NNp with a total of 5000 weights and NNm with 600 weights in it. Further details can be found in Modi et al. (2018). The position network takes in non-local features as a flattened array that can also be implemented as a convolution kernel of size 3 in TensorFlow.

The results for `FlowPM` reconstruction are shown in Figure 15 and 16 and the results are consistent with the `FastPM` reconstruction shown in Modi et al. (2018). We are primarily interested in the cross-correlation of the reconstructed field with the true initial field and for that we improve over the standard reconstruction methods (Eisenstein et al., 2007) which have cross-correlation of ~90% at k=0.2 h/Mpc (Modi et al., 2018). Though `FlowPM` reconstructed field has more power on small scales than the true field or `FastPM` reconstruction, this can be corrected with simulations as pointed out in Seljak et al. (2017); Modi et al. (2018); Horowitz et al. (2019b) but it is beyond the scope and also tangential to the main point of this work.

BFGS) algorithm (Nocedal, 1980) which improves the `FastPM` reconstruction slightly with a little increase in wall clock of time, up to 4.8 seconds per iteration.

```python
def recon_model(mesh, data, x0, M0):
    # [...] some code initialisation
    # Load pre-trained variables of NN
    # Define initial conditions as variable
    var = mtf.get_variable(mesh, 'linear',
            shape, tf.constant_initializer(x0))
    # Forward model here with FlowPM
    final_field = FLOWPM(var, ...)
    # Supplement FlowPM with NN models
    position = NNp(field_field)
    mass = NNm(final_field)
    model = position*mass
    #Define metrics for loss with data noise variance
    #"sigma" and prior power spectrum "power"
    chisq = mtf.reduce_sum(
            ((mu + mtf.log(model+M0) -
            mtf.log(data+M0))/sigma)^2)
    prior = mtf.reduce_sum(r2c(var)^2/power)
    loss = chisq + prior
    return var, model, loss

def model_fn(ft, data, mode, params):
    # [...] some code initialisation
    var, model, loss = recon_model(mesh, data,
            ft['x0'], ft['M0'])
    # same as Figure 12 from here

def main():
    # [...] some code initialisation
    # Estimator with dict input
    def input_fn():
        ft = {'x0':x0, 'M0':M0}
        return ft, data
    # same as Figure 12 from here
    recon_estimator = tf.estimator.Estimator(
            model_fn=model_fn,model_dir=./tmp/)
    # Train (Reconstruct)
    recon_estimator.train(input_fn=input_fn,
            max_steps=100)
    # and evaluate model.
    eval_results = recon_estimator.predict(
            input_fn=input_fn)["ic"]
```

Figure 14: (Listing 7.2) Update Listing 11 to include neural networks in the forward model and associated variables to be changed over iterations of reconstruction

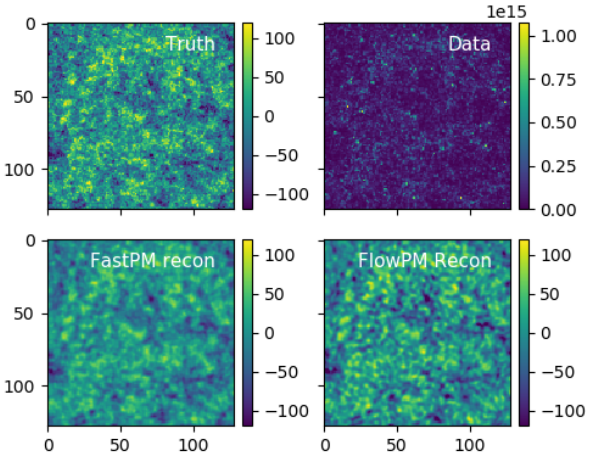

Figure 15: Reconstructing initial conditions from the discrete halo mass field observable. Top row shows the true initial conditions (left) and the halo mass field (right). Bottom row shows the reconstructed MAP initial density field with `FastPM` and `FlowPM` respectively.

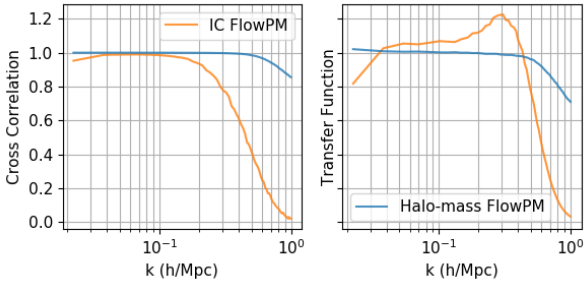

Figure 16: Two point statistics for the reconstructed fields from the halo mass field observable. We show the cross correlation coefficient (left) and transfer function (right) for the reconstructed initial field (orange) and the corresponding halo mass field (blue) with `FlowPM`.

In addition to the ease of implementing reconstruction in FlowPM, the most significant gain of `FlowPM` is in terms of the speed of iteration. The time taken for 1 `FlowPM` iteration is $\sim$ 1.7 second while the time taken for 1 iteration in `FastPM` reconstruction in $\sim$ 15 seconds. The huge increase in time for `FastPM` iterations as compared to `FlowPM` is due to the Python implementation of single convolution kernel and its gradients as required by the neural network bias model, which is very efficiently implemented in TensorFlow. Due to increased complexity in the forward model, gradient descent with line search does not converge at all. Thus for `FastPM` reconstruction, we use L-BFGS optimization with line search in `abopt` package. Since it is a pseudo second order scheme, it outperforms gradient descent and Adam with the optimization converging in 450 iterations. Since Mesh-TF does not have L-BFGS optimization implemented, `FlowPM` implementation uses Adam algorithm as before and does 100 iterations at every annealing step, totaling to 1500 iterations. Despite the difference in the number of iterations, `FlowPM` implementation takes only 1 hour on a single GPU while `FastPM` reconstruction takes roughly 2 hours on 128 CPUs (MPI processes).

8. Conclusion

In this work, we present `FlowPM` - a cosmological N-body code implemented in Mesh-TensorFlow for GPU-accelerated, distributed, and differentiable simulations to tackle the unprecedented modeling and analytic challenges posed by the next generation of cosmological surveys.

`FlowPM` implements `FastPM` scheme as the underlying particle-mesh gravity solver, with gravitational force estimated with 3D Fourier transforms. For distributed computation, we use Mesh-TensorFlow as our model parallelism framework which gives us full flexibility in determining the computational layout and distribution of our simulation. To overcome the bottleneck of large scale distributed 3D FFTs, we propose and implement a novel multi-grid force computation scheme based on image pyramids. We demonstrate that without any fine tuning, this method is able to achieve sub-percent accuracy while

reducing the communication data-volume by a factor of 64x. At the same time, given the GPU accelerations, `FlowPM` is 4-20x faster than the corresponding Python `FastPM` simulation depending on the resolution and compute distribution.

However the main advantage of `FlowPM` is the differentiability of the simulation and natural interfacing with deep learning frameworks. Built entirely in TensorFlow, the simulations are differentiable with respect to every component. This allows for development of novel analytic methods such as simulations based inference and reconstruction of cosmological fields which rely on being able to analytically estimate the response (gradients) of the cosmological observables (evolved dark matter field for proxy) with respect to the input cosmology and underlying initial density field, especially in the large data volumes and multi-million dimensionality of cosmological problems. We demonstrate with two examples, providing the logical code-snippets, how `FlowPM` makes the latter straightforward by using TensorFlow Estimator API to do optimization in 128^3 dimensional space. It is also able to naturally interface with machine learning frameworks as a part of the forward model in this reconstruction. Lastly, due to its speed, it is able to achieve comparable accuracy to `FastPM` based reconstruction in 5 times lower wall-clock time (550 seconds vs 2300 seconds), and 640 times lower computing time (single GPU vs 128 processes) for dark matter field observables.

`FlowPM` is open-source and publicly available on our Github repo at https://github.com/modichirag/flowpm. We provide example code to do forward PM evolution with single-grid and multi-grid force computation. We also provide code for the reconstruction with both the examples demonstrated in the paper. We hope that this will encourage the community to use this novel tool and develop scientific methods to tackle the next generation of cosmological surveys.

Acknowledgements

We would like to thank Mustafa Mustafa and Wahid Bhimji at NERSC and Thiru Palanisamy at Google for the support in developing this. We would also like to acknowledge the gracious donation of TPU compute time from Google which was instrumental in developing and testing `FlowPM` for SIMD Mesh implementation. This material is based upon work supported by the National Science Foundation under Grant Numbers 1814370 and NSF 1839217, and by NASA under Grant Number 80NSSC18K1274. This research used resources of the National Energy Research Scientific Computing Center (NERSC), a U.S. Department of Energy Office of Science User Facility operated under Contract No. DE-AC02-05CH11231.

References

Alsing, J., Wandelt, B., Feeney, S., 2018. Massive optimal data compression and density estimation for scalable, likelihood-free inference in cosmology. MNRAS 477, 2874–2885. doi:10.1093/mnras/sty819, arXiv:1801.01497.

Anderson, C.H., Bergen, J.R., Burt, P.J., Ogden, J.M., 1984. Pyramid methods in image processing.

Burt, P., Adelson, E., 1983. The laplacian pyramid as a compact image code. IEEE Transactions on Communications 31, 532–540.

Cheng, H.T., Haque, Z., Hong, L., Ispir, M., Mewald, C., Polosukhin, I., Roumpos, G., Sculley, D., Smith, J., Soergel, D., Tang, Y., Tucker, P., Wicke, M., Xia, C., Xie, J., 2017. TensorFlow Estimators: Managing Simplicity vs. Flexibility in High-Level Machine Learning Frameworks. arXiv e-prints , arXiv:1708.02637arXiv:1708.02637.

Colavincenzo, M., Sefusatti, E., Monaco, P., Blot, L., Crocce, M., Lippich, M., Sánchez, A.G., Alvarez, M.A., Agrawal, A., Avila, S., Balaguera-Antolínez, A., Bond, R., Codis, S., Dalla Vecchia, C., Dorta, A., Fosalba, P., Izard, A., Kitaura, F.S., Pellejero-Ibanez, M., Stein, G., Vakili, M., Yepes, G., 2018. Comparing approximate methods for mock catalogues and covariance matrices – III: bispectrum. Monthly Notices of the Royal Astronomical Society 482, 4883–4905. URL: https://doi.org/10.1093/mnras/sty2964, doi:10.1093/mnras/sty2964, arXiv:https://academic.oup.com/mnras/article-pdf/482/4/4883/268222

Cranmer, K., Brehmer, J., Louppe, G., 2019. The frontier of simulation-based inference. arXiv e-prints , arXiv:1911.01429arXiv:1911.01429.

DESI Collaboration, Aghamousa, A., Aguilar, J., Ahlen, S., Alam, S., Allen, L.E., Allende Prieto, C., Annis, J., Bailey, S., Balland, C., et al., 2016. The DESI Experiment Part I: Science,Targeting, and Survey Design. ArXiv e-prints arXiv:1611.00036.

Eisenstein, D.J., Seo, H.J., Sirko, E., Spergel, D.N., 2007. Improving Cosmological Distance Measurements by Reconstruction of the Baryon Acoustic Peak. ApJ 664, 675–679. doi:10.1086/518712, arXiv:astro-ph/0604362.

Elsner, F., Schmidt, F., Jasche, J., Lavaux, G., Nguyen, N.M., 2019. Cosmology Inference from Biased Tracers using the EFT-based Likelihood. arXiv e-prints , arXiv:1906.07143arXiv:1906.07143.

Feng, Y., Chu, M.Y., Seljak, U., McDonald, P., 2016. FASTPM: a new scheme for fast simulations of dark matter and haloes. MNRAS 463, 2273–2286. doi:10.1093/mnras/stw2123, arXiv:1603.00476.

Feng, Y., Seljak, U., Zaldarriaga, M., 2018. Exploring the posterior surface of the large scale structure reconstruction. Journal of Cosmology and Astro-Particle Physics 2018, 043. doi:10.1088/1475-7516/2018/07/043, arXiv:1804.09687.

Garrison, L.H., Eisenstein, D.J., Ferrer, D., Tinker, J.L., Pinto, P.A., Weinberg, D.H., 2018. The Abacus Cosmos: A Suite of Cosmological N-body Simulations. ApJS 236, 43. doi:10.3847/1538-4365/aabfd3, arXiv:1712.05768.

Greengard, L., Rokhlin, V., 1987. A Fast Algorithm for Particle Simulations. Journal of Computational Physics 73, 325–348. doi:10.1016/0021-9991(87)90140-9.

Harnois-Déraps, J., Pen, U.L., Iliev, I.T., Merz, H., Emberson, J.D., Desjacques, V., 2013. High-performance P^3M N-body code: CUBEP^{3}M. MNRAS 436, 540–559. doi:10.1093/mnras/stt1591, arXiv:1208.5098.

He, S., Li, Y., Feng, Y., Ho, S., Ravanbakhsh, S., Chen, W., Póczos, B., 2019. Learning to predict the cosmological structure formation. Proceedings of the National Academy of Science 116, 13825–13832. doi:10.1073/pnas.1821458116, arXiv:1811.06533.

Hockney, R.W., Eastwood, J.W., 1988. Computer Simulation Using Particles. Taylor & Francis, Inc., Bristol, PA, USA.

Horowitz, B., Lee, K.G., White, M., Krolewski, A., Ata, M., 2019a. TARDIS. I. A Constrained Reconstruction Approach to Modeling the z ∼ 2.5 Cosmic Web Probed by Lyα Forest Tomography. ApJ 887, 61. doi:10.3847/1538-4357/ab4d4c, arXiv:1903.09049.

Horowitz, B., Seljak, U., Aslanyan, G., 2019b. Efficient optimal reconstruction of linear fields and band-powers from cosmological data. J. Cosmology Astropart. Phys. 2019, 035. doi:10.1088/1475-7516/2019/10/035, arXiv:1810.00503.

Izard, A., Crocce, M., Fosalba, P., 2016. ICE-COLA: towards fast and accurate synthetic galaxy catalogues optimizing a quasi-N-body method. MNRAS 459, 2327–2341. doi:10.1093/mnras/stw797, arXiv:1509.04685.

Jasche, J., Wandelt, B.D., 2013. Bayesian physical reconstruction of initial conditions from large-scale structure surveys. MNRAS 432, 894–913. doi:10.1093/mnras/stt449, arXiv:1203.3639.

Kingma, D.P., Ba, J., 2014. Adam: A Method for Stochastic Optimization. ArXiv e-prints arXiv:1412.6980.

Kitaura, F.S., Yepes, G., Prada, F., 2014. Modelling baryon acoustic oscillations with perturbation theory and stochastic halo biasing. MNRAS 439, L21–L25. doi:10.1093/mnrasl/slt172, arXiv:1307.3285.

Kodi Ramanah, D., Charnock, T., Villaescusa-Navarro, F., Wandelt, B.D., 2020. Super-resolution emulator of cosmological simulations using deep physical models. arXiv e-prints , arXiv:2001.05519arXiv:2001.05519.

LSST Science Collaboration, Abell, P.A., Allison, J., Anderson, S.F., Andrew, J.R., Angel, J.R.P., Armus, L., Arnett, D., Asztalos, S.J., Axelrod, T.S., et al., 2009. LSST Science Book, Version 2.0. ArXiv e-prints arXiv:0912.0201.

Matsubara, T., 2008. Nonlinear perturbation theory with halo bias and redshift-space distortions via the Lagrangian picture. Phys. Rev. D 78, 083519. doi:10.1103/PhysRevD.78.083519, arXiv:0807.1733.

Merz, H., Pen, U.L., Trac, H., 2005. Towards optimal parallel PM N-body codes: PMFAST. New A 10, 393–407. doi:10.1016/j.newast.2005.02.001, arXiv:astro-ph/0402443.

Modi, C., Castorina, E., Feng, Y., White, M., 2019a. Intensity mapping with neutral hydrogen and the Hidden Valley simulations. arXiv e-prints , arXiv:1904.11923arXiv:1904.11923.

Modi, C., Feng, Y., Seljak, U., 2018. Cosmological reconstruction from galaxy light: neural network based light-matter connection. Journal of Cosmology and Astro-Particle Physics 10, 028. doi:10.1088/1475-7516/2018/10/028, arXiv:1805.02247.

Modi, C., White, M., Slosar, A., Castorina, E., 2019b. Reconstructing large-scale structure with neutral hydrogen surveys. arXiv e-prints , arXiv:1907.02330arXiv:1907.02330.

Monaco, P., 2016. Approximate methods for the generation of dark matter halo catalogs in the age of precision cosmology. Galaxies 4. URL: https://www.mdpi.com/2075-4434/4/4/53, doi:10.3390/galaxies4040053.

Nelson, D., Pillepich, A., Genel, S., Vogelsberger, M., Springel, V., Torrey, P., Rodriguez-Gomez, V., Sijacki, D., Snyder, G.F., Griffen, B., Marinacci, F., Blecha, L., Sales, L., Xu, D., Hernquist, L., 2015. The illustris simulation: Public data release. Astronomy and Computing 13, 12–37. doi:10.1016/j.ascom.2015.09.003, arXiv:1504.00362.

Nocedal, J., 1980. Updating quasi-newton matrices with limited storage. Mathematics of Computation 35, 773–782. URL: http://www.jstor.org/stable/2006193.

Pippig, M., 2013. Pfft - an extension of fftw to massively parallel architectures. SIAM Journal on Scientific Computing 35, C213 – C236. doi:10.1137/120885887.

Quinn, T., Katz, N., Stadel, J., Lake, G., 1997. Time stepping N-body simulations. arXiv e-prints , astro–ph/9710043arXiv:astro-ph/9710043.

Schmittfull, M., Simonović, M., Assassi, V., Zaldarriaga, M., 2018. Modeling Biased Tracers at the Field Level. arXiv e-prints arXiv:1811.10640.

Seljak, U., Aslanyan, G., Feng, Y., Modi, C., 2017. Towards optimal extraction of cosmological information from nonlinear data. Journal of Cosmology and Astro-Particle Physics 2017, 009. doi:10.1088/1475-7516/2017/12/009, arXiv:1706.06645.

Shazeer, N., Cheng, Y., Parmar, N., Tran, D., Vaswani, A., Koanantakool, P., Hawkins, P., Lee, H., Hong, M., Young, C., Sepassi, R., Hechtman, B.A., 2018. Mesh-tensorflow: Deep learning for supercomputers. CoRR abs/1811.02084. URL: http://arxiv.org/abs/1811.02084, arXiv:1811.02084.

Sheth, R.K., Tormen, G., 1999. Large-scale bias and the peak background split. MNRAS 308, 119–126. doi:10.1046/j.1365-8711.1999.02692.x, arXiv:astro-ph/9901122.

Springel, V., Pakmor, R., Zier, O., Reinecke, M., 2020. Simulating cosmic structure formation with the GADGET-4 code. arXiv e-prints , arXiv:2010.03567arXiv:2010.03567.

Stein, G., Alvarez, M.A., Bond, J.R., 2019. The mass-Peak Patch algorithm for fast generation of deep all-sky dark matter halo catalogues and its N-body validation. MNRAS 483, 2236–2250. doi:10.1093/mnras/sty3226, arXiv:1810.07727.

Suisalu, I., Saar, E., 1995. An adaptive multigrid solver for high-resolution cosmological simulations. MNRAS 274, 287–299. doi:10.1093/mnras/274.1.287, arXiv:astro-ph/9412043.

Tassev, S., Zaldarriaga, M., Eisenstein, D.J., 2013. Solving large scale structure in ten easy steps with COLA. J. Cosmology Astropart. Phys. 2013, 036. doi:10.1088/1475-7516/2013/06/036, arXiv:1301.0322.

Teyssier, R., 2002. Cosmological hydrodynamics with adaptive mesh refinement. A new high resolution code called RAMSES. A&A 385, 337–364. doi:10.1051/0004-6361:20011817, arXiv:astro-ph/0111367.

Wang, H., Mo, H.J., Yang, X., Jing, Y.P., Lin, W.P., 2014. ELUCID—Exploring the Local Universe with the Reconstructed Initial Density Field. I. Hamiltonian Markov Chain Monte Carlo Method with Particle Mesh Dynamics. ApJ 794, 94. doi:10.1088/0004-637X/794/1/94, arXiv:1407.3451.

White, M., Tinker, J.L., McBride, C.K., 2014. Mock galaxy catalogues using the quick particle mesh method.

Zel'Dovich, Y.B., 1970. Reprint of 1970A&A.....5...84Z. Gravitational instability: an approximate theory for large density perturbations. A&A 500, 13–18.

Highlights

- End-to-end differentiable cosmological N-Body simulations
- GPU based simulations with 10x speed gain over current CPU simulations
- First N-body simulation written in TensorFlow interfacing with ML and DL components
- Novel multi-grid force algorithm for distributed computing of large scale forces.
- Support for large-scale distribution on supercomputers with Mesh TensorFlow

Author Contribution

Chirag Modi: Conceptualization; Formal analysis; Investigation; Methodology; Software; Validation; Visualization; Roles/Writing - original draft; **Francois Lanusse:** Conceptualization; Formal analysis; Investigation; Methodology; Resources; Software; Supervision; Validation; Writing - review & editing **Uros Seljak:** Conceptualization; Project administration; Resources; Supervision; Writing - review & editing

Declaration of interests

X The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

☐The authors declare the following financial interests/personal relationships which may be considered as potential competing interests: